Karlsruhe Institute of Technology

Engineering von Modularity-basiertem Graphenclustern

Studienarbeit
von

Geraud Oscar Fofie Lafou

An der Fakultät für Informatik
Institut für Theoretische Informatik

Gutachter:	Prof. Dr. Dorothea Wagner
Betreuende Mitarbeiterin:	Dipl.-Inf. Andrea Schumm
Betreuender Mitarbeiter:	Dr.rer.nat. Robert Görke

Bearbeitungszeit: 30. April 2010 – 22. September 2010

Inhaltsverzeichnis

1. Einleitung

Diese Arbeit behandelt das Clustern von Graphen. Bei diesem Problem geht es darum, den Graphen so in Teilgraphen zu partitionieren, dass es möglichst viele Kanten innerhalb der entstehenden Teilgraphen und möglichst wenige Kanten zwischen ihnen gibt. Basierend auf dieser Intuition können Graphen nach verschiedenen formellen Kriterien geclustert werden. Grundsätzlich werden hier Graphen so partitioniert, dass beliebig viele Teilgraphen entstehen und jeder entstehende Teilgraph eine beliebige Anzahl von Knoten enthalten kann. Diese Teilgraphen sollen knotendisjunkt sein. Es werden sowohl zusammenhängende als auch unzusammenhängende Graphen geclustert. Es werden ungewichtete Graphen behandelt.

Graphenclustern, auch genannt Clustering, wird vor allem in sozialen Netzwerken angewendet, die zum Beispiel politische Zusammenhänge, wissenschaftliche Kollaborationen in der Forschungsliteratur, Kommunikationsnetze, Abhängigkeiten in der Ökonomie, Proteininteraktion in Organismen, Räuber-Beute-Beziehungen in Ökosystemen, Freundschaften bei Facebook, studi-vz, usw. repräsentieren.

Beim Graphenclustern werden die entstehenden Teilgraphen Cluster und die dazugehörige Partition der Knotenmenge Clusterung genannt. Die Qualität einer Clusterung des Graphen kann mit verschiedenen Funktionen bewertet werden. Solche Funktionen quantifizieren, wie groß bzw. wie klein die Dichte der Kanten innerhalb Clustern bzw. zwischen ihnen ist. Diese Arbeit beschäftigt sich mit einer speziellen Qualitätsfunktion namens Modularity. Modularity bewertet jede Clusterung mit einer Zahl, so dass gute Clusterungen große Werte bekommen. Die Maximierung dieser Funktion ist ein NP-vollständiges Problem. Für dieses Problem wird oft ein Greedy-Algorithmus namens *CNM Algorithmus* verwendet. Der Algorithmus baut die Clusterung iterativ aus einzelnen Knoten zusammen, so dass sich der Wert von Modularity bei jedem Schritt erhöht. Der Algorithmus bricht ab, wenn es kein Paar von Clustern mehr im Graphen gibt, dessen Vereinigung eine Erhöhung von Modularity bringt. Oft gibt es bei jedem Schritt im Algorithmus mehrere Clusterpaare, die einen Zuwachs an Modularity bringen. Dann wird das Clusterpaar, das vereinigt wird, entsprechend einer Funktion, der sogenannten Prioritätsfunktion, gewählt.

In dieser Arbeit werden zuerst die hier angesprochenen Begriffe Clustering und Modularity definiert und der CNM Algorithmus wird beschrieben. Dann stellen wir einige Prioritätsfunktionen aus der Literatur vor. Danach stellen wir die von uns entwickelten eigenen Prioritätsfunktionen vor. Anschließend kombinieren wir verschiedene Prioritätsfunktionen mit dem Originalalgorithmus. Bei den Kombinationen werden Cluster bis zu einer bestimmten Anzahl von Schritten nach einer beliebigen Prioritätsfunktion vereinigt. Dann

werden sie nach der Prioritätsfunktion des Originalalgorithmus vereinigt. Die Idee hinter dieser Kombination ist, dass zu Beginn des Algorithmus frühe, ungüngstige Entscheidungen vermieden werden und später die Maximierung von Modularity in Vordergrund steht. Zum Schluss vergleichen wir die verschiedenen Prioritätsfunktionen und die Ergebnisse der Kombinationen.

2. Grundlagen

2.1. Definitionen

In dieser Arbeit bezeichnen wir mit $G = (V, E)$ einen einfachen, ungerichteten Graphen ohne Mehrfachkanten und Schleifen, mit $n = |V|$ die Anzahl der Knoten von G und mit $m = |E|$ die Anzahl der Kanten von G. $\mathcal{C} = \{C_1, \ldots, C_k\}$ ist eine Partition von V, wobei $1 \leq k \leq n$ und die C_i nichtleere Mengen sind. Wir nennen $\mathcal{C}$ eine *Clusterung* von G und die C_i *Cluster*. Cluster C_i bezeichnet auch den knoteninduzierten Subgraphen von G, $G[C_i] := (C_i, E(C_i))$, mit $E(C_i) := \{\{v, w\} \in E : v, w \in C_i\}$. Die Menge $E(\mathcal{C}) := \bigcup_{i=1}^{k} E(C_i)$ ist die Menge der *Intraclusterkanten*. $E \setminus E(\mathcal{C})$ ist die Menge der *Interclusterkanten* und $E(C_i, C_j)$ ist die Menge der Kanten, die C_i und C_j verbinden. $m(\mathcal{C})$ ist die Anzahl der Intraclusterkanten und $\overline{m}(\mathcal{C})$ ist die Anzahl der Interclusterkanten.

2.1.1. Clustering und einfache Qualitätsfunktionen

Clustering ist ein Verfahren, um eine Clusterung eines Graphen zu finden, so dass die Dichte von Kanten innerhalb von Clustern größer als die zwischen diesen Clustern ist. Die Anzahl aller möglichen Clusterungen eines Graphen ist exponentiell in seiner Größe. Damit man *gute* Clusterungen von *schlechten* Clusterungen unterscheiden kann, ist es notwendig, ein quantitatives Kriterium festzulegen, um die Güte zu messen. Eine *Qualitätsfunktion* ist eine Funktion, die jeder Clusterung eine Zahl zuordnet. So werden Clusterungen nach den Werten ihrer Qualitätsfunktion bewertet: Clusterungen mit höherer Zahl sind *gut* und die mit der größten Zahl ist die *optimale Clusterung* des Graphen.

Es gibt mehrere Qualitätsfunktionen, unter anderem *Coverage*: Coverage ist das Verhältnis der Anzahl der Intraclusterkanten (Kanten innerhalb von Clustern) zu der Anzahl der gesamten Kanten.

$$cov(\mathcal{C}) = \sum_{C \in \mathcal{C}} \frac{|E(C)|}{m} = \frac{m(\mathcal{C})}{m}.$$

Eine andere Qualitätsfunktion ist *Performance*: Performance zählt die Anzahl von Knotenpaaren, die zu einem Cluster gehören und adjazent sind bzw. die Anzahl von Knotenpaaren, die zu unterschiedlichen Clustern gehören und nicht adjazent sind. Diese Anzahl wird dann durch die Anzahl der gesamten Kanten in einem vollständigen Graphen mit n Knoten geteilt. Performance wird wie folgt berechnet:

$$P(\mathcal{C}) = \frac{|\{\{i,j\} \in E, C_i = C_j\}| + |\{\{i,j\} \notin E, C_i \neq C_j\}|}{\binom{n}{2}}$$
$$= \frac{|\{\{i,j\} \in E, C_i = C_j\}| + |\{\{i,j\} \notin E, C_i \neq C_j\}|}{n(n-1)/2},$$

wobei C_i Cluster des Knoten i ist.

Die am meisten verwendete Qualitätsfunktion ist *Modularity* von Newman und Girvan.

2.1.2. Modularity

Die Idee hinter Modularity ist, für eine gegebene Clusterung $\mathcal{C}=\{C_1, \ldots, C_k\}$ des Graphen den Anteil der Kanten in Clustern mit dem Erwartungswert der Kanten in den Subgraphen $C_1, \ldots, C_k$ in einem Zufallsmodell zu vergleichen. Sinnvolle Clusterungen des Graphen sind Clusterungen, bei denen es eine Variation zwischen dem Anteil der Kanten in Clustern und dem Erwartungswert der Kanten in den entsprechenden Clustern im Zufallsmodell gibt. Clusterungen mit einem großen Anteil der Kanten in Clustern haben einen hohen Wert bezüglich Modularity.

Das *Nullmodell* ist ein Zufallsmodell, nach dem die Kanten zufällig in den Graphen eingefügt werden. Dafür gibt es mehrere Möglichkeiten. Wir nutzen ein Nullmodell, das den Erwartungswert der Knotengrade und die Anzahl der gesamten Kanten und Knoten im Graphen beibehält. Es gibt m Runden. Bei jeder Runde bildet sich eine Kante: Entweder werden zwei beliebige verschiedene Knoten nacheinander gewählt, oder ein Knoten wird zweimal gewählt. Jeder Knoten v kann höchstens $2m$ Male jeweils mit der Wahrscheinlichkeit $p_v = deg(v)/2m$ gewählt werden. Der Zufallsgraph hat also insgesamt m Kanten.

Wir berechnen schrittweise die Modularity einer gegebenen Clusterung $\mathcal{C}=\{C_1, \ldots, C_k\}$. Dafür berechnen wir für jedes Cluster C_i den Anteil der Intraclusterkanten und die erwartete Coverage im entsprechenden Subgraphen des Zufallsgraphen.

Sei $\mathbb{A}_i$ der Anteil der Kanten innerhalb des Clusters C_i:

$$\mathbb{A}_i = \frac{|E(C_i)|}{m}.$$

Im Subgraphen C_i des Nullmodells entsteht eine ungerichtete Kante beim Knoten u bzw. zwischen verschiedenen Knoten v und w, indem bei einer Runde u zweimal gewählt wird bzw. zuerst v gewählt wird und dann w oder umgekehrt. Der erwartete Anteil der Kanten innerhalb des Subgraphen innerhalb des Subgraphen C_i im Nullmodell ist der Erwartungswert $\mathbb{E}_i$:

$$\mathbb{E}_i = \mathbb{E}\left(\frac{|E(C_i)|}{m}\right) = m \cdot \frac{\sum_{v,w \in C_i} p_v p_w}{m} = \sum_{v,w \in C_i} \frac{deg(v)}{2m}\frac{deg(w)}{2m} = \sum_{v,w \in C_i} \frac{deg(v)deg(w)}{(2m)^2}$$
$$= \frac{1}{(2m)^2} \sum_{v,w \in C_i} deg(v)deg(w) = \frac{1}{(2m)^2}\left[\sum_{v \in C_i} deg(v)\right]^2 = \left[\frac{\sum_{v \in C_i} deg(v)}{2m}\right]^2.$$

Die Modularity der Clusterung $\mathcal{C}=\{C_1, \ldots, C_k\}$ ist die Summe der Differenz $\mathbb{A}_i - \mathbb{E}_i$ über alle Cluster und wird wie folgt definiert:

$$q(\mathcal{C}) = cov(\mathcal{C}) - \mathbb{E}(cov(\mathcal{C})) = \sum_{C \in \mathcal{C}} \left[\frac{|E(C)|}{m} - \left(\frac{\sum_{v \in C} deg(v)}{2m}\right)^2\right].$$

Der minimale Wert von Modularity ist -1/2. Dies ergibt sich bei der Clusterung $\mathcal{C} = \{C_a, C_b\}$ eines bipartiten Graphen $K_{a,b}$. In diesem Fall ist $\sum_{v \in C} deg(v)$ gleich m und $\frac{|E(C)|}{m}$ null (es gibt keine Intraclusterkante). Im Spezialfall des Graphen ohne Kanten ist Coverage per Definition 1. So hat jede Clusterung dieses Graphen Modularity 1. Im Allgemeinen gilt: $-1/2 \leq q(\mathcal{C}) \leq 1$ [BDH+07]. Modularity ist ein quantitatives Maß für die Qualität einer Clusterung und wird oft benutzt, um die Qualität von verschiedenen Clusterungen eines Graphen zu vergleichen. Clusterungen mit großen Werten an Modularity sind gute Clusterungen. Also wollen wir beim Clustering Modularity maximieren, um gute Clusterungen zu bekommen.

Es ist bewiesen worden, dass die Optimierung von Modularity ein NP-vollständiges Problem ist (*Brandes et al.*) [BDH+07]. Es gibt jedoch viele Heuristiken, die ziemlich gute Werte für Modularity in relativ guter Laufzeit finden. Oft wird ein Greedy-Algorithmus verwendet, um eine Clusterung des Graphen mit einem hohen Wert an Modularity zu finden. Wir stellen im nächsten Abschnitt zuerst den naiven Greedy-Algorithmus, dann die verbesserte Version von *Aaron Clauset*, *M. E. J. Newman*, *Cristopher Moore* (CNM Algorithmus) [CNM04] vor.

2.2. Greedy-Algorithmus

Der Greedy-Algorithmus vereinigt iterativ zwei Cluster, die den größten Zuwachs an Modularity bringen. Der Algorithmus beginnt mit der Clusterung $\mathcal{C} = \{C_1, \ldots, C_n\}$. Dabei enthält Cluster $C_i = \{v_i\}$ genau einen Knoten v_i. Der Anteil der Knotengrade im Cluster C_i ist $a_i = \sum_{v \in C_i} deg(v)/2m$ und $e_{ij} = |E(C_i, C_j)|/m$ bezeichnet den Anteil der Kanten, die Cluster C_i und C_j verbinden. $\mathcal{C}_{i,j}$ ist die Clusterung, die sich aus $\mathcal{C}$ ergibt, wenn Cluster C_i und C_j vereinigt werden und alle andere Cluster unverändert bleiben. Das heißt, $\mathcal{C}_{i,j} = (\mathcal{C} \setminus \{C_i, C_j\}) \bigcup \{C_i \cup C_j\}$. Die Differenz $\Delta_{i,j} = q(\mathcal{C}_{i,j}) - q(\mathcal{C})$ ist die Erhöhung bzw. die Verringerung der Modularity zwischen Clusterung $\mathcal{C}$ und Clusterung $\mathcal{C}_{i,j}$.

$$
\begin{aligned}
\Delta_{i,j} &= q(\mathcal{C}_{i,j}) - q(\mathcal{C}) \\
&= \sum_{C \in \mathcal{C}_{i,j}} \left[\frac{|E(C)|}{m} - \left(\frac{\sum_{v \in C} deg(v)}{2m} \right)^2 \right] - \sum_{C \in \mathcal{C}} \left[\frac{|E(C)|}{m} - \left(\frac{\sum_{v \in C} deg(v)}{2m} \right)^2 \right] \\
&= \sum_{C_k \in \mathcal{C}, k \neq i,j} \left[\frac{|E(C_k)|}{m} - \left(\frac{\sum_{v \in C_k} deg(v)}{2m} \right)^2 \right] + \frac{|E(C_i \cup C_j)|}{m} - \left(\frac{\sum_{v \in C_i \cup C_j} deg(v)}{2m} \right)^2 \\
&\quad - \left[\sum_{C_k \in \mathcal{C}, k \neq i,j} \left[\frac{|E(C_k)|}{m} - \left(\frac{\sum_{v \in C_k} deg(v)}{2m} \right)^2 \right] + \frac{|E(C_i)|}{m} - \left(\frac{\sum_{v \in C_i} deg(v)}{2m} \right)^2 \right. \\
&\quad \left. + \frac{|E(C_j)|}{m} - \left(\frac{\sum_{v \in C_j} deg(v)}{2m} \right)^2 \right] \\
&= \frac{|E(C_i \cup C_j)| - |E(C_i)| - |E(C_j)|}{m} - \left(\frac{\sum_{v \in C_i} deg(v) + \sum_{v \in C_j} deg(v)}{2m} \right)^2 \\
&\quad + \left(\frac{\sum_{v \in C_i} deg(v)}{2m} \right)^2 + \left(\frac{\sum_{v \in C_j} deg(v)}{2m} \right)^2 \\
&= \frac{|E(C_i, C_j)|}{m} - \frac{\sum_{v \in C_i} deg(v) \sum_{v \in C_j} deg(v)}{2m^2} \\
\Delta_{i,j} &= e_{ij} - 2a_i a_j. \qquad (2.1)
\end{aligned}
$$

2.2.1. Erste Version des Algorithmus

Die Werte $\Delta_{i,j}$ sind die Einträge einer Matrix Δ. Für jedes Clusterpaar (C_i, C_j) wird der Wert $\Delta_{i,j}$ berechnet. Das Clusterpaar (C_i, C_j), dessen $\Delta_{i,j}$ der größte Wert ist, wird zu einem Cluster vereinigt, so dass die Clusterung $\mathcal{C}_{i,j}$ einen hohen Wert an Modularity hat. Nach der Vereinigung von zwei Clustern C_i und C_j wird das Verfahren mit der neuen Clusterung $\mathcal{C}_{i,j}$ weiter ausgeführt. So werden iterativ zwei Cluster nach jedem Schritt vereinigt. Dabei werden die Einträge der Matrix Δ aktualisiert. Nach Formel 2.1 werden nur Einträge $\Delta_{i,k}$, $\Delta_{k,i}$, $\Delta_{j,k}$, $\Delta_{k,j}$, $1 \leq k \leq n$ geändert: Alle anderen Einträge bleiben unverändert. Wir nennen das neue Cluster $C_i \cup C_j$ wieder C_i, aktualisieren den Wert a_i und löschen die Zeile j und die Spalte j.

Das Verfahren endet, wenn es keinen positiven Eintrag mehr in der Matrix gibt. In diesem Fall kann der Wert an Modularity nicht mehr erhöht werden. Ansonsten werden immer zwei Cluster zu einem vereinigt, so dass die neue Clusterung immer ein Cluster weniger enthält. Der Algorithmus endet in diesem Fall nach $n-1$ Schritten ($|\mathcal{C}| = 1$). *Algorithm* 1 stellt den Pseudocode des oben beschriebenen Verfahrens vor. In den Zeilen 3 bis 15 werden die Einträge $\Delta_{i,j}$, e_{ij} und a_i initialisiert.

Algorithm 1: Greedy algorithm for maximizing modularity

```
1  Input: a simple graph G = (V, E)
2  Output: a clustering C of G with high modularity
3  Initialization: C ← {C_i = {v_i} : v_i ∈ V}
4  for all pairs C_i, C_j ∈ C do
5      if {i, j} ∈ E then
6          set e_ij ← 1/m
7      else
8          set e_ij ← 0
9  for C_i ∈ C do
10     deg_i := 0
11     for C_j ∈ C do
12         deg_i := deg_i + e_ij · m
13     set a_i ← deg_i / 2m
14 for all pairs C_i, C_j ∈ C do
15     set Δ_{i,j} ← e_ij − 2 a_i a_j
16 while |C| > 1 and max(Δ) > 0 do
17     find i, j with Δ_{i,j} = max(Δ)
18     set C ← C_{i,j} := (C \ {C_i, C_j}) ⋃ {C_i ∪ C_j}
19     for C_k ∈ C do
20         Δ_{i,k} ← Δ_{i,k} + Δ_{j,k}
21         Δ_{k,i} ← Δ_{k,i} + Δ_{k,j}
22     delete the jth row and the jth column of the matrix Δ
23 return C
```

Der Algorithmus endet nach höchstens $n-1$ Schritten. Bei jedem Schritt des Algorithmus wird der maximale Wert $\Delta_{i,j}$ von Δ gesucht. Danach werden die Werte $\Delta_{i,k}$ für alle Cluster C_k berechnet, die mindestens mit einem der Cluster C_i, C_j verbunden sind. Es gibt höchstens $n-2$ solche Cluster. Nach Formel 2.1 hat die Matrix Δ nicht-positive Einträge für alle Clusterpaare, die mit keiner Kante verbunden sind. Das positive Maximum von Δ wird nur unter den Einträgen $\Delta_{i,j}$ für Clusterpaare (C_i, C_j) gesucht, die mindestens mit einer Kante verbunden sind ($|E(C_i, C_j)| > 0$). Es gibt höchstens m solche Paare. So kann das Maximum $\max(\Delta)$ in $O(m)$ gefunden werden. Die Gesamtlaufzeit ist in $O(n(m+n))$

und der Speicheraufwand ist in $O(n^2)$.

2.2.2. Verbesserte Datenstrukturen

Um eine bessere Laufzeit des Algorithmus zu erzielen und den Bedarf an Speicherplatz zu reduzieren, haben *M. E. J. Newman*, *Aaron Clauset* und *Cristopher Moore* den Algorithmus mit besseren Datenstrukturen implementiert. Die Matrix Δ wird wie folgt konstruiert: Jede Zeile ist ein binärer Heap der Länge maximal n. Das Maximum jeder Zeile wird in einem Max-Heap der Länge maximal n gespeichert. So ist der größte Wert der Matrix Δ das Maximum des Max-Heap, das sich besser berechnen lässt.

Der Wert von Modularity kann nur bei Vereinigung von zusammenhängenden Clustern erhöht werden (Formel 2.1). So werden in jeder Zeile i der Matrix Δ nur Werte $\Delta_{i,j}$ für Clusterpaare (C_i, C_j) eingetragen, die mindestens mit einer Kante verbunden sind $(e_{ij} > 0)$. Wenn zwei Cluster C_i und C_j vereinigt werden, dann werden im Heap H_i der Zeile i die Einträge $\Delta_{i,k}$ aktualisiert und die Einträge $\Delta_{j,l}$ des Heaps H_j werden aktualisiert und im Heap H_i eingefügt, falls der Eintrag $\Delta_{i,l}$ nicht in H_i vorhanden ist. Danach wird der Heap H_j entfernt. Alle Einträge $\Delta_{k,i}$ und $\Delta_{k,j}$ werden auch aktualisiert. Dabei werden die Einträge $\Delta_{k,j}$ nach der Aktualisierung gelöscht oder durch $\Delta_{k,i}$ ersetzt, falls kein Eintrag $\Delta_{k,i}$ auf dem Heap H_k vorhanden ist. Nach der Vereinigung $C_i \cup C_j := C_i$ werden die neuen Einträge bei der Aktualisierung wie folgt berechnet: Wir nutzen die Formel 2.1.

$$\begin{aligned} \Delta_{i,k} :&= (e_{ik} + e_{jk}) - 2\,(a_i + a_j)\,a_k \\ &= (e_{ik} - 2a_i a_k) + (e_{jk} - 2a_j a_k)\,. \end{aligned} \tag{2.2}$$

Wir machen eine Fallunterscheidung danach, wie Cluster C_k mit den Clustern C_i und C_j verbunden ist.

- Fall 1: Cluster C_k ist mit den beiden Clustern C_i und C_j verbunden, dann gilt: $e_{ik} > 0$, $e_{jk} > 0$ und $\Delta_{i,k} := \Delta_{i,k} + \Delta_{j,k}$; $\Delta_{k,i} := \Delta_{k,i} + \Delta_{k,j}$.
- Fall 2: Cluster C_k ist nur mit einem Cluster verbunden. Sei ohne Beschränkung der Allgemeinheit C_i dieses Cluster, dann gilt: $e_{jk} = 0$ und $\Delta_{i,k} := \Delta_{i,k} - 2a_j a_k$; $\Delta_{k,i} := \Delta_{k,i} - 2a_k a_j$.

Diese verbesserte Version vom ersten Algorithmus wird in *Algorithm* 2 dargestellt. Die Initialisierung von e_{ij}, a_i ist wie in *Algorithm* 1.

2.2.3. Grobe Laufzeitanalyse

Wir berechnen die Laufzeit des Algorithmus mit diesen Datenstrukturen. Nach der Vereinigung der Cluster C_i und C_j wird die Zeile j gelöscht und die Zeile i aktualisiert. Sei α_r die Anzahl der Cluster mit denen Cluster C_r verbunden ist. Im Heap H_i werden höchstens $\alpha_i + \alpha_j$ Einträge aktualisiert oder eingefügt. Das ergibt einen Aufwand von $O((\alpha_i + \alpha_j) \log n)$. In jedem Heap H_k mit $k \neq i, j$ werden höchstens zwei Werte aktualisiert oder gelöscht. Es gibt höchstens $\alpha_i + \alpha_j$ Heaps, deren Einträge aktualisiert werden. Es werden also höchstens $\alpha_i + \alpha_j$ Einträge im Max-Heap geändert. Das Maximum aller Maxima wird also in $O((\alpha_i + \alpha_j) \log n)$ gefunden. Jedes Cluster C_i ist mit höchstens $n - 1$ Clustern verbunden, daraus folgt $\alpha_i \leq n - 1$ und $\alpha_i + \alpha_j < 2n$. Der Algorithmus endet nach höchstens $n - 1$ Schritten mit einer Gesamtlaufzeit in $O(n^2 \log n)$.

Die Laufzeit des Algorithmus lässt sich noch anders berechnen. Wir beschreiben im nächsten Abschnitt diese andere Analyse. Dafür definieren wir zuerst den Begriff Dendrogramm.

Algorithm 2: Greedy algorithm for maximizing modularity : CNM algorithm

Input: a simple graph $G = (V, E)$
Output: a clustering $\mathcal{C}$ of G with high modularity
$\mathcal{C} \leftarrow \{C_i = \{v_i\} : v_i \in V\}$
initialize e_{ij}, a_i
for *all pairs* $C_i, C_j \in \mathcal{C}$ *with* $e_{ij} > 0$ **do**
    set $\Delta_{i,j} \leftarrow e_{ij} - 2a_i a_j$
while $|\mathcal{C}| > 1$ *and* $\max(\Delta) > 0$ **do**
    find i, j with $\Delta_{i,j} = \max(\Delta)$
    set $\mathcal{C} \leftarrow \mathcal{C}_{i,j} := (\mathcal{C} \setminus \{C_i, C_j\}) \bigcup \{C_i \cup C_j\}$
    for *all connected pairs* $C_i, C_k \in \mathcal{C}$ **do**
        if $e_{ik} > 0$ *and* $e_{jk} > 0$ **then**
            $\Delta_{i,k} \leftarrow \Delta_{i,k} + \Delta_{j,k}$
            $\Delta_{k,i} \leftarrow \Delta_{k,i} + \Delta_{k,j}$
            $e_{ik} \leftarrow e_{ik} + e_{jk}$
            delete $\Delta_{k,j}$
        else
            if $e_{ik} > 0$ **then**
                $\Delta_{i,k} \leftarrow \Delta_{i,k} - 2a_j a_k$
                $\Delta_{k,i} \leftarrow \Delta_{k,i} - 2a_k a_j$
            else
                $\Delta_{i,k} \leftarrow \Delta_{j,k} - 2a_i a_k$
                $\Delta_{k,i} \leftarrow \Delta_{k,j} - 2a_i a_k$
                $e_{ik} \leftarrow e_{jk}$
                delete $\Delta_{k,j}$
        $a_i \leftarrow a_i + a_j$
    delete Heap H_j
return $\mathcal{C}$

2.3. Dendrogramm und Laufzeitanalyse des Greedy-Algorithmus

Bei der Ausführung des Greedy-Algorithmus lässt sich ein binärer Baum konstruieren. Es werden iterativ zwei Cluster C_i und C_j zu einem Cluster $C_{i,j} := C_i \cup C_j$ vereinigt. Dabei bildet sich bei jedem Schritt des Algorithmus ein Baumknoten $C_{i,j}$ mit zwei Kindern C_i und C_j. Angenommen der Greedy-Algorithmus wird bis zu $n-1$ Schritten ausgeführt. Das heißt, das zweite Abbruchkriterium ($\max(\Delta) > 0$) wird nicht berücksichtigt. Der binäre Baum, der sich bei der Ausführung des Greedy-Algorithmus bildet, heißt *Dendrogramm.*

Beispiel 1 *Sei $G = (V, E)$ ein Graph mit $V = \{1, 2, 3, 4, 5\}$, und $C_1 = \{1\}, C_2 = \{2\}, C_3 = \{3\}, C_4 = \{4\}, C_5 = \{5\}$ die Cluster bei der Initialisierung des Algorithmus. Es haben sich im Laufe des Algorithmus schrittweise folgende Cluster geformt.*

- *Schritt 1:* $C_{1,2} := C_1 \cup C_2$
- *Schritt 2:* $C_{3,4} := C_3 \cup C_4$
- *Schritt 3:* $C_{1,2,3,4} := C_{1,2} \cup C_{3,4}$
- *Schritt 4:* $C_{1,2,3,4,5} := C_{1,2,3,4} \cup C_5$

Abbildung 2.1 zeigt das Dendrogramm, das nach der Ausführung des Greedy-Algorithmus für den gegebenen Graphen konstruiert wird.

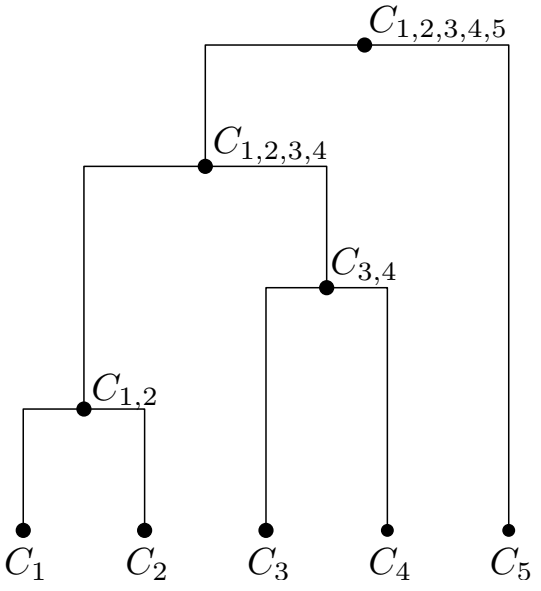

Abbildung 2.1.: Beispiel eines Dendrogramms.

Wir bezeichnen mit d die Anzahl der Kanten, die auf dem längsten Pfad von der Wurzel zu einem Blatt des Dendrogramms liegen. Jeder Knoten v trägt bei jeder Vereinigung, an der er beteiligt ist, höchstens $2 \cdot deg(v)$ Heap-Operationen bei. Der Knoten ist an höchstens d Vereinigungen beteiligt. Die Anzahl der Heap-Operationen bei den Vereinigungen sind also höchstens:

$$\sum_{v \in V} d \cdot deg(v) = d \cdot \sum_{v \in V} deg(v) = d \cdot 2m.$$

Die Kosten für die Aktualisierung eines Heapeintrags betragen $O(\log n)$. Die gesamten Kosten im Algorithmus sind also in $O(md \log n)$.

Um die Kosten zu reduzieren, ist es erwünscht das Dendrogramm zu balancieren, damit jeder Knoten an möglichst wenigen Vereinigungen beteiligt ist.

Der CNM Algorithmus ergibt aber kein balanciertes Dendrogramm. Abbildung 2.2 und Abbildung 2.3 zeigen einen Graphen und das entsprechende Dendrogramm, das sich nach der Ausführung des CNM Algorithmus ergibt.

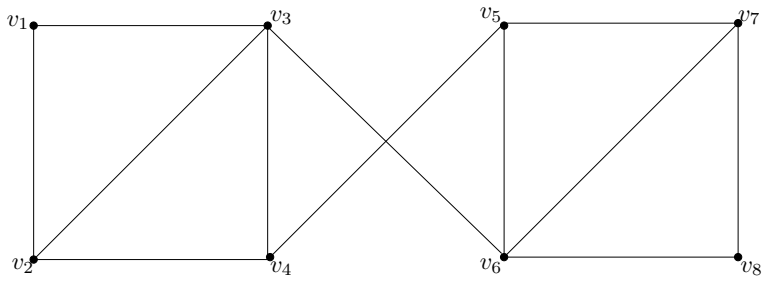

Abbildung 2.2.: Beispiel eines Graphen mit unbalanciertem Dendrogramm bei der Ausführung des CNM Algorithmus.

Bei der Ausführung des CNM Algorithmus für diesen Graphen haben sich die Cluster $C_1 := v_{1,2,3,4} := \{v_1, v_2, v_3, v_4\}$ und $C_2 := v_{5,6,7,8} := \{v_5, v_6, v_7, v_8\}$ gebildet. Abbildung 2.3 zeigt das Dendrogramm, das nach der Ausführung konstruiert wird.

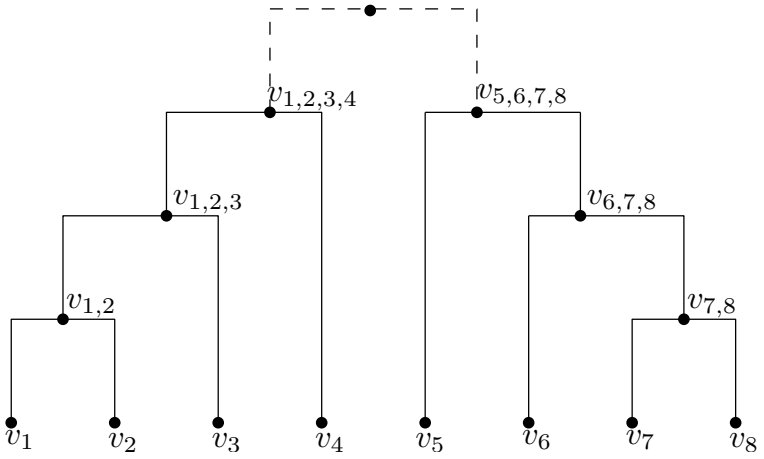

Abbildung 2.3.: Unbalanciertes Dendrogramm des Graphen in Abbildung 2.2 bei der Ausführung des CNM Algorithmus. Die Höhe des Dendrogramms ist $d = 4$.

Für dieselbe Clusterung dieses Graphen könnten die gleichen Cluster in einer anderen Reihenfolge erzeugt werden. Abbildung 2.4 zeigt ein Beispiel für ein balanciertes Dendrogramm dieser Clusterung.

Wir bemerken, dass der CNM Algorithmus ein unbalanciertes Dendrogramm erzeugt. Mehrere Autoren haben deshalb den Algorithmus so modifiziert, dass das Dendrogramm balanciert wird. Wir stellen im nächsten Kapitel einige balancierte Versionen des CNM Algorithmus vor.

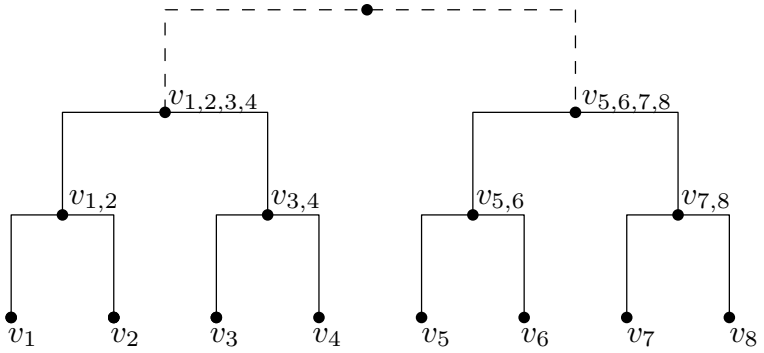

Abbildung 2.4.: Beispiel eines balancierten Dendrogramms des Graphen in Abbildung 2.2. Die Höhe dieses Dendrogramms ist $d = 3$.

3. Balancierte Versionen des CNM-Algorithmus

Mit einem balancierten Dendrogramm wird der CNM Algorithmus effizienter. Dafür haben mehrere Autoren verschiedene Modifikationen des Algorithmus vorgestellt. In diesen Versionen wird das Dendrogramm dadurch balanciert, dass im Algorithmus nicht immer das Clusterpaar mit dem größten Zuwachs an Modularity vereinigt wird, sondern ein anderes Clusterpaar, das nach einer *Prioritätsfunktion* gewählt wird. Eine Prioritätsfunktion ist eine Funktion, die jedem Clusterpaar eine Zahl zuordnet: die *Priorität*. Clusterpaare mit größerer Priorität werden bei der Vereinigung vor anderen Clusterpaaren bevorzugt.

Wir stellen in diesem Kapitel den Ansatz von *Ken Wakita* und *Toshiyuki Tsurumi* [WT07], die Prioritätsfunktionen von *Andreas Noack* und *Randolf Rotta* [NR09], die von *Danon et al.* [DDGA06] und unsere eigenen Prioritätsfunktionen vor. Als weitere Heuristik haben wir alle Prioritätsfunktionen mit dem Originalalgorithmus kombiniert. Diese Kombinationen werden am Ende des Kapitels vorgestellt.

3.1. Ansatz von Ken Wakita und Toshiyuki Tsurumi

Ken Wakita und *Toshiyuki Tsurumi* sind die ersten, die modifizierte Prioritätsfunktionen verwendet haben, um das Dendrogramm zu balancieren. Dazu stellen Sie drei Heuristiken vor, die auf einer Funktion *ratio* basieren.

Ebenfalls ändern die Autoren die Datenstrukturen, um aufwändige Operationen wie zum Beispiel die Suche des Clusterpaars $\{i, j\}$ mit dem maximalen Wert $\Delta_{i,j}$ und die Vereinigung von zwei Clustern schneller berechnen zu können. Die binären Heaps im CNM Algorithmus werden durch doppelt verkettete Listen ersetzt. Jedes Cluster C_i hat einen Zeiger auf seine adjazenten Cluster, die in einer doppelt verketteten Liste gespeichert sind. Die Schlüssel dieser Liste sind Clusterpaare (C_i, C_j), wobei C_j ein zu C_i adjazentes Cluster ist. Jede Liste $L(C_i)$ ist nach der ID der zu C_i adjazenten Cluster sortiert. Die Autoren geben an, zwei Cluster können dadurch in Linearzeit vereinigt werden.

Um den größten Wert $\max(\Delta)$ aller Werte $\Delta_{i,j}$ der Clusterpaare (C_i, C_j) zu finden, wird für jedes Cluster ein Zeiger *"max Δ is"* gespeichert, der in seiner Liste auf das Clusterpaar mit dem größten Wert $\Delta_{i,j}$ verweist. Der größte Wert $\Delta_{i,j}$ jeder Liste wird in einen Max Heap eingetragen. Bei dieser Vorgehensweise gibt es ein Probem bei der Aktualisierung der Listen. Wenn zwei Cluster C_i und C_j vereinigt worden sind und der Wert $\Delta_{k,i}$ der

Liste eines Clusters C_k aktualisiert worden ist, muss der Zeiger *"max Δ is"* der Liste des Clusters C_k auch aktualisiert werden. Dafür müssen alle Clusterpaare der Liste des Clusters C_k durchlaufen werden. Für die Behebung des Problems wird der Zeiger *"max Δ is"* bei der Aktualisierung der Werte $\Delta_{k,i}$ in der Liste des Clusters C_k nach Fallunterscheidung gleichzeitig aktualisiert.

- Fall 1: Wenn (C_i, C_k) nicht das Clusterpaar ist, auf das *"max Δ is"* verweist, und der Wert $\Delta_{i,k}$ sich verringert, dann wird *"max Δ is"* nicht geändert.
- Fall 2: Wenn (C_i, C_k) nicht das Clusterpaar ist, auf das *"max Δ is"* verweist, und der Wert $\Delta_{i,k}$ sich erhöht, dann wird dieser Wert mit dem entsprechenden Wert des Clusters, auf das *"max Δ is"* verweist, verglichen. Wenn der Wert $\Delta_{i,k}$ größer ist, dann wird *"max Δ is"* auf das Clusterpaar (C_i, C_k) umgebogen.
- Fall 3: Wenn (C_i, C_k) das Clusterpaar ist, auf das *"max Δ is"* verweist, und der Wert $\Delta_{i,k}$ sich erhöht, dann wird *"max Δ is"* nicht geändert.
- Fall 4: Wenn (C_i, C_k) das Clusterpaar ist, auf das *"max Δ is"* verweist, und der Wert $\Delta_{i,k}$ sich verringert, dann müssen alle Clusterpaare der Liste durchlaufen werden, um *"max Δ is"* zu aktualisieren.

Ken Wakita und *Toshiyuki Tsurumi* behaupten, dass der letzte Fall wegen eines Prozesses namens *Preferential Attachment* selten vorkomme. Preferential Attachment ist ein Prozess, nach dem sich soziale Netzwerke bilden: Bei ihrer Verbindung zu anderen Knoten im Netz bevorzugen neue Knoten die wichtigsten Knoten (Knoten mit höherem Grad) vor anderen Knoten. In ähnlicher Weise verhält sich der Zuwachs an Modularity beim CNM Algorithmus. Wenn der Zeiger *"max Δ is"* im (q-1)-ten Schritt auf ein Clusterpaar (C_k, C_i) verweist, dann verweist er im q-tem Schritt mit großen Wahrscheinlichkeit immer noch auf das Clusterpaar (C_k, C_i). Die Autoren nehmen an, dass Fall 4 deshalb selten vorkomme und das Clusterpaar $\{i, j\}$ mit dem maximalen Wert $\Delta_{i,j}$ wegen dieser Verhaltensweise in $O(1)$ gefunden werde. Dieses Verhalten sollte aber bei einem balancierten Dendrogramm nicht vorkommen. Deshalb kann in diesem Fall der Aufwand, um das Maximum zu finden nicht so einfach durch $O(1)$ abgeschätzt werden.

Im Algorithmus wird eine Funktion

$$ratio(C_i, C_j) = \min\left(\frac{size(C_i)}{size(C_j)}, \frac{size(C_j)}{size(C_i)}\right)$$

benutzt. In jedem Schritt wird das Clusterpaar (C_i, C_j) vereinigt, für das

$$\Delta_{i,j} \cdot ratio(C_i, C_j)$$

maximal ist. Es werden also Clusterpaare vereinigt, die fast gleich groß sind.

Ken Wakita und *Toshiyuki Tsurumi* stellen drei Heuristiken vor, um die Effizienz des Algorithmus zu verbessern. Die Heuristiken unterscheiden sich in der Berechnung der Größe $size(C_i)$ eines Clusters C_i.

- Die erste Heuristik *(HE)* berechnet die Größe eines Clusters C_i als die Anzahl seiner adjazenten Cluster, d.h. die Anzahl der Clusterpaare in seiner Liste. Diese Heuristik wird benutzt, weil die Kosten der Vereinigung von Clustern proportional zu der Anzahl ihrer Clusterpaare sind.
- Die zweite Heuristik *(HE')* berechnet zuerst für jede Liste das Maximum $\Delta_{i,j}$ wie beim CNM Algorithmus. Dann wird ratio(C_i, C_j) für das entsprechende Clusterpaar (C_i, C_j) wie bei (HE) berechnet. Im globalen Heap wird $\Delta_{i,j}\cdot$ ratio(C_i, C_j) benutzt, um das Maximum zu berechnen.
- Die dritte Heuristik *(HN)* berechnet die Größe eines Clusters als die Anzahl der Knoten in diesem Cluster.

3.2. Andere Prioritätsfunktionen

Andreas Noack und *Randolf Rotta* haben den CNM Algorithmus anders modifiziert [NR09]. Das Papier ist sehr neu und lag bei der Aufgabenstellung der Studienarbeit noch nicht vor. Die dabei verwendete Multi-Level-Technik geht über das Thema der Studienarbeit hinaus. Aus diesem Grund beschränken wir uns in diesem Abschnitt auf die von den Autoren benutzten Prioritätsfunktionen.

Die Autoren haben drei verschiedene Prioritätsfunktionen benutzt:

- Die Prioritätsfunktion *WD: Weight Density* ist gleich

$$\frac{\Delta_{i,j}}{deg(C_i)deg(C_j)},$$

  wobei $deg(C_k) = \sum_{v \in C_k} deg(v)$ für ein Cluster C_k. Diese Prioritätsfunktion berechnet bis auf einen konstanten Faktor das Verhältnis des Gewichtes der Kanten zwischen zwei Clustern zu dem Erwartungswert des Gewichtes dieser Kanten im Nullmodell.

- Die Prioritätsfunktion *Sig: Significance* ist gleich

$$\frac{\Delta_{i,j}}{\sqrt{deg(C_i)deg(C_j)}}.$$

  Dabei ist $deg(C_i)$ wie oben definiert. Dies ist ein Kompromiss zwischen dem Zuwachs an Modularity und Weight Density. Die Autoren betrachten den Wert $\frac{deg(C_i)deg(C_j)}{deg(V)}$ als die Varianz des Kantengewichtes zwischen C_i und C_j für große Werte $deg(V)$. Daraus folgt, dass die Standardabweichung der Kantengewichte zwischen C_i und C_j bis auf einen konstanten Faktor gleich $\sqrt{deg(C_i)deg(C_j)}$ ist. Das heißt, Significance ist die tatsächliche Abweichung des Kantengewichtes vom Erwartungswert geteilt durch die Standardabweichung.

- Eine andere Prioritätsfunktion, die die Autoren benutzt haben, ist die von *Danon et al.* und ist gleich

$$\frac{\Delta_{i,j}}{\min\left(deg(C_i), deg(C_j)\right)}.$$

  Der Wert $deg(C_i)$ hat die gleiche Bedeutung wie vorher. Diese Prioritätsfunktion wurde vorgeschlagen, damit große Cluster nicht mehr wie im CNM Algorithmus bevorzugt werden [DDGA06].

3.3. Eigene Prioritätsfunktionen

Wie wir schon bemerkt haben, wird der Algorithmus wegen der Unbalanciertheit des Dendrogramms beim CNM Algorithmus verlangsamt. Ähnlich wie bei Ken Wakita und Toshiyuki Tsurumi wollen wir die Berechnung verschnellern. Dafür stellen wir in diesem Abschnitt andere Heuristiken vor.

Die Idee ist, bei der Vereinigung von Clustern diejenigen mit wenigen ausgehenden Kanten zu bevorzugen. Bei der Vereinigung von zwei Clustern hängt die Anzahl der für die Aktualisierung benötigten Heap-Operationen von der Anzahl der adjazenten Cluster ab. Die Vereinigung von Clusterpaaren (C_i, C_j), so dass $C_i \cup C_j$ eine kleine Anzahl ausgehender Kanten hat, verursacht weniger Kantenaktualisierungen, deshalb werden solche Clusterpaare bevorzugt. Auf diese Weise wollen wir die Laufzeit verbessern. Zusätzlich wollen wir für eine gute Qualität immer wenige Interclusterkanten haben. Auch deshalb wird die Entstehung von Clustern mit weniger ausgehenden Kanten bevorzugt. Darüber hinaus

werden große Cluster nicht mehr wie im CNM Algorithmus bevorzugt. Das Ziel dabei ist, den CNM Algorithmus so zu modifizieren, dass das Dendrogramm balancierter wird.

Im Greedy-Algorithmus werden iterativ zwei Cluster vereinigt, die großen Zuwachs an Modularity bringen und eine kleine Anzahl ausgehender Kanten haben. Sei $\mathcal{E}(C_i)$ der *externe Grad* eines Clusters C_i: Das heißt, die Anzahl ausgehender Kanten dieses Clusters.

$$\begin{aligned}\mathcal{E}(C_i) :&= |\{u,v\} \in E, u \in C_i \wedge v \notin C_i| \\ &= \sum_{v \in C_i} deg(v) - 2\,|E(C_i)| \\ &= 2m\left(\frac{\sum_{v \in C_i} deg(v)}{2m} - \frac{|E(C_i)|}{m}\right) \\ &= 2m\,(a_i - \mathbb{A}_i)\end{aligned}$$

Bei der Vereinigung werden Clusterpaare (C_i, C_j) mit einem kleinen Wert $\mathcal{E}(C_i \cup C_j)$ bevorzugt.

$$\begin{aligned}\mathcal{E}(C_i \cup C_j) &= \mathcal{E}(C_i) + \mathcal{E}(C_j) - 2m \cdot e_{ij} \\ &= 2m\,(a_i - \mathbb{A}_i + a_j - \mathbb{A}_j - e_{ij}) \\ &= 2m\,[(a_i + a_j) - (\mathbb{A}_i + \mathbb{A}_j) - e_{ij}]\,. \end{aligned} \tag{3.1}$$

Wir vereinigen das Clusterpaar (C_i, C_j), für das $\Delta_{i,j}/\mathcal{E}(C_i \cup C_j)$ maximal ist. Diese neue Prioritätsfunktion nennen wir *$\mathcal{E}$-Priorität*. Sie ist wie folgt definiert:

$$\Delta'_{i,j} = \frac{\Delta_{i,j}}{\mathcal{E}(C_i \cup C_j)}$$

Falls $\mathcal{E}(C_i \cup C_j)$ gleich Null ist, dann wird $\Delta'_{i,j}$ auf $\Delta_{i,j}$ gesetzt.

Die Anzahl $\mathcal{E}(C_i \cup C_j)$ der ausgehenden Kanten des Clusters $C_i \cup C_j$ ist eine obere Schranke für die Anzahl der benötigten Heap-Operationen bei der Vereinigung von Clustern C_i und C_j. Als weitere Prioritätsfunktion haben wir in der obigen Prioritätsfunktion den Wert $\mathcal{E}(C_i \cup C_j)$ durch einen kleineren Wert $\sqrt{\mathcal{E}(C_i \cup C_j)}$ ersetzt. Der Wert $\sqrt{\mathcal{E}(C_i \cup C_j)}$ wächst langsamer als $\mathcal{E}(C_i \cup C_j)$. Diese Prioritätsfunktion heißt *$\mathcal{E}$-schwache Priorität* und ist gleich

$$\frac{\Delta_{i,j}}{\sqrt{\mathcal{E}(C_i \cup C_j)}}.$$

$\mathcal{E}$-schwachePriorität ist ein Kompromiss zwischen dem Originalalgorithmus und *$\mathcal{E}$-Priorität*.

Wir stellen jetzt eine andere Prioritätsfunktion vor. Wir bezeichnen mit $\mathcal{A}(C_i)$ der *abstrakte Grad* eines Clusters C_i: Das heißt, die Anzahl der zu C_i adjazenten Cluster. Bei der Vereinigung von zwei Clustern C_i und C_j werden $\mathcal{A}(C_i) + \mathcal{A}(C_j)$ Heap-Operationen benötigt. Um die Gesamtzahl der Kantenaktualisierungen zu verringern wollen wir Clusterpaare (C_i, C_j) mit einem kleinen Wert $\mathcal{A}(C_i) + \mathcal{A}(C_j)$ bevorzugen. Diese Prioritätsfunktion heißt *$\mathcal{A}$-Priorität* und ist gleich

$$\frac{\Delta_{i,j}}{\mathcal{A}(C_i) + \mathcal{A}(C_j)}.$$

Analog wie oben, definieren wir ein Kompromiss zwischen dem Originalalgorithmus und *$\mathcal{A}$-Priorität*. Diese Prioritätsfunktion heißt *$\mathcal{A}$-schwache Priorität* und ist gleich

$$\frac{\Delta_{i,j}}{\sqrt{\mathcal{A}(C_i) + \mathcal{A}(C_j)}}.$$

3.4. Kombination verschiedener Prioritätsfunktionen

Als eine andere Heuristik, um das Dendrogramm des CNM Algorithmus zu balancieren, können verschiedene Prioritätsfunktionen kombiniert werden. Bis zu einem bestimmten Zeitpunkt im Algorithmus wird eine der in diesem Kapitel vorgestellte Prioritätsfunktionen benutzt. Dann wird wie im Originalalgorithmus $\Delta_{i,j}$ benutzt.

Die Idee dahinter ist, dass wir vermuten, dass der CNM Algorithmus vor allem am Anfang unbalanciert arbeitet.

Für einen gegebenen Graphen $G = (V, E)$ mit $|V| = n$ wird eine balancierte Version des Algorithmus solange ausgeführt, bis $\lceil (1 - l) \cdot n \rceil$ Cluster von den n Clustern am Anfang verbleiben. Dabei ist $l \in \mathbb{R}$, $0 \leq l \leq 1$ ein Tuning Parameter. Dann wird der CNM Algorithmus mit den verbleibenden Clustern und dem Zuwachs an Modularity als Prioritätsfunktion solange ausgeführt, bis es kein Clusterpaar mehr mit einem Zuwachs an Modularity gibt.

Wir haben diese Kombinationen mit verschiedenen Werten l ausgewertet: $l = 0$, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, $\cdots$, 0.95, 1.

Die Ergebnisse der Experimente mit den in diesem Kapitel vorgestellten Prioritätsfunktionen können im Kapitel 5 angeschaut werden. Zuerst stellen wir im nächsten Kapitel die Datenstrukturen vor, mit denen wir die Algorithmen implementiert haben.

4. Datenstrukturen und Implementierung

Im ersten Teil dieses Kapitels wird die für die Implementierung der vorgestellten Heuristiken benutzte Datenstruktur vorgestellt. Bei allen Heuristiken wird die Prioritätsfunktion nach jeder Vereinigung von Clusterpaaren aktualisiert. Dies ist das Thema des zweiten Teils dieses Kapitels.

4.1. Die Datenstruktur

Die Datenstruktur, die wir für die Implementierung des CNM Algorithmus und der verschiedenen Versionen benutzt haben, besteht aus einem globalem Max-Heap, Vektoren und verketteten Listen.

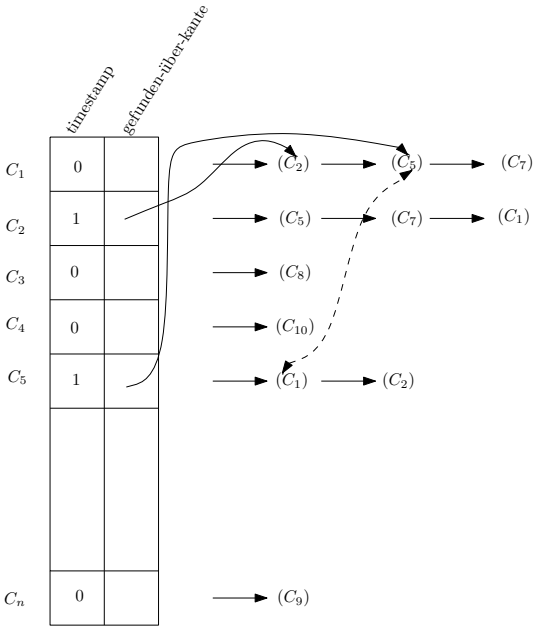

Abbildung 4.1.: Illustration der Datenstruktur.

Zu jedem Cluster werden die zu ihm adjazenten Cluster als verkettete Liste gespeichert. Jedes Cluster hat ein Attribut *timestamp* und einen Zeiger *"gefunden-über-kante"*. Zusätzlich gibt es eine Variable *time*, die mit 1 initialisiert wird. Es gilt am Anfang: timestamp(C_i):=0

für alle Cluster C_i. Jedes adjazente Clusterpaar (C_i, C_j) wird als eine Kante gespeichert. Jede Kante $e{:=}\{C_i, C_j\}$ hat eine eindeutige *ID* und zwei Gewichte e_{ij} und $\Delta_{i,j}$. Zur Illustration der Datenstruktur siehe Abbildung 4.1.

Um zwei Cluster C_i und C_j zu vereinigen, werden zuerst die Cluster gesucht, die zu C_i und C_j adjazent sind. Dafür wird zuerst über alle Nachbarkanten $e{=}\{C_i, C_k\}$ von C_i iteriert. Für die Cluster C_k werden timestamp(C_k):= *time* und *"gefunden-über-kante"*$(C_k){:=}e$ gesetzt. Dann wird über alle Nachbarkanten $e'{=}\{C_j, C_l\}$ von C_j iteriert. Wenn timestamp(C_l) gleich *time* ist, dann ist C_l ein gemeinsamer Nachbar und $e{:=}$*"gefunden-über-kante"*(C_l) und $e'{=}\{C_i, C_k\}$ sind die dazugehörige Kanten. Die Gewichte der Kante e werden aktualisiert und die Kante e' wird gelöscht.

Alle zu C_j adjazenten Clustern werden an die Liste C_i angehängt. Die Gewichte der anderen zu C_i oder C_j adjazenten Kanten werden auch aktualisiert. Der Wert von *time* wird um eins erhöht. Bei der Vereinigung von zwei Clustern C_i und C_j wird über C_i und C_j iteriert. Dies entspricht dem Aufwand von $O\left(deg(C_i) + deg(C_j)\right)$. Die Vereinigung von Clustern ohne den Aufwand für die Heap-Operationen ist also linear in der Anzahl der zu C_i und C_j adjazenten Cluster.

Um den Wert $\max(\Delta)$ zu finden, werden alle Werte $\Delta_{i,j}$ in einem Max-Heap der Länge m gespeichert. Eine Heap-Operation ist in $O(\log m) \subseteq O(\log n^2) = O(2\log n) = O(\log n)$. Nachdem ein maximaler Wert $\Delta_{i,j}$ gefunden wird, wird er im Heap gelöscht. Die anderen Heapeinträge werden wie folgt aktualisiert:

- Falls sich zwei Einträge $\Delta_{i,k}$ und $\Delta_{j,k}$ im Heap befinden, dann wird der Eintrag $\Delta_{i,k}$ aktualisiert und $\Delta_{j,k}$ wird gelöscht.
- Falls sich nur der Eintrag $\Delta_{i,k}$ im Heap befinden, dann wird er aktualisiert.
- Falls sich nur der Eintrag $\Delta_{j,k}$ im Heap befinden, dann wird er aktualisiert und durch den Eintrag $\Delta_{i,k}$ ersetzt.

Der Gesamtaufwand für die Vereinigung von zwei Clustern C_i und C_j ist also in $O((deg(C_i)+ deg(C_j)) \cdot \log n)$.

4.2. Aktualisierung der Prioritätsfunktionen

In diesem Abschnitt wird beschrieben, wie Prioritätsfunktionen bei der Implementierung aktualisiert werden. Wir haben als Beispiel die Aktualisierung unserer eigenen Prioritätsfunktion erläutert. Diese Aktualisierung wird auch im Pseudocode unserer balancierten Version vorgestellt. Die Aktualisierung anderer Prioritätsfunktionen ist analog.

Wenn zwei Cluster C_i und C_j vereinigt werden, dann werden die Einträge $\Delta'_{i,k}$ und $\Delta'_{j,k}$ aktualisiert. Das neue Cluster $C_i \cup C_j$ heißt C_i. Die neuen Werte sind $\Delta'_{i,k} = \Delta_{i,k}/\mathcal{E}(C_i \cup C_k)$. Der Wert $\Delta_{i,k}$ wird wie im CNM Algorithmus nach Formel 2.2 berechnet. Die Werte $\mathcal{E}\left(C_i \cup C_k\right)$ werden wie folgt berechnet:

$$\begin{aligned}
\mathcal{E}\left(C_i \cup C_k\right) &\overset{3.1}{=} 2m\left[\left((a_i + a_j) + a_k\right) - \left((\mathbb{A}_i + \mathbb{A}_j + e_{ij}) + \mathbb{A}_k\right) - (e_{ik} + e_{jk})\right] \\
&= 2m\left[\left((a_i + a_j) - (\mathbb{A}_i + \mathbb{A}_j) - e_{ij}\right)\right] + 2m\left(a_k - \mathbb{A}_k - e_{ik} - e_{jk}\right) \\
&\overset{3.1}{=} \mathcal{E}\left(C_i \cup C_j\right) + 2m\left(a_k - \mathbb{A}_k - e_{ik} - e_{jk}\right).
\end{aligned}$$

Wir machen eine Fallunterscheidung danach, wie Cluster C_k mit den Clustern C_i und C_j verbunden ist. Der Lesbarkeit halber werden die Werte $\mathcal{E}\left(C_i \cup C_j\right)$ mit $\mathcal{E}_{ij}$ bezeichnet.

- Fall 1: Cluster C_k ist mit den beiden Clustern C_i und C_j verbunden, dann gilt: $e_{ik} > 0$, $e_{jk} > 0$ und

$$\Delta_{i,k}^{'} = \frac{\Delta_{i,k} + \Delta_{j,k}}{\mathcal{E}_{ik}}$$

mit

$$\mathcal{E}_{ik} = \mathcal{E}_{ij} + 2m\left(a_k - \mathbb{A}_k - e_{ik} - e_{jk}\right).$$

- Fall 2: Cluster C_k ist nur mit einem Cluster verbunden. Sei ohne Beschränkung der Allgemeinheit C_i dieses Cluster, dann gilt: $e_{jk} = 0$ und

$$\Delta_{i,k}^{'} = \frac{\Delta_{i,k} - 2a_j a_k}{\mathcal{E}_{ik}}$$

mit

$$\mathcal{E}_{ik} = \mathcal{E}_{ij} + 2m\left(a_k - \mathbb{A}_k - e_{ik}\right).$$

Diese modifizierte Version des CNM Algorithmus wird in *Algorithm 3* dargestellt. Wir speichern für jedes Cluster C_i ein neues Attribut $\mathbb{A}_i$ für den Anteil der Kanten innerhalb des Clusters, das mit 0 initialisiert wird. Die Initialisierungen und die Aktualisierungen der anderen Werten sind wie im CNM Algorithmus.

Algorithm 3: Balanced Greedy algorithm for maximizing modularity

Input: a simple graph $G = (V, E)$
Output: a clustering $\mathcal{C}$ of G with high modularity
$\mathcal{C} \leftarrow \{C_i = \{v_i\} : v_i \in V\}$
initialize a_i, $\mathbb{A}_i$, e_{ij}, $\Delta_{i,j}$, $\mathcal{E}_{ij}$ and $\Delta_{i,j}^{'}$ for all connected pairs $C_i, C_j \in \mathcal{C}$
while $|\mathcal{C}| > 1$ *and* $\max(\Delta^{'}) > 0$ **do**
    find i, j with $\Delta_{i,j}^{'} = \max(\Delta^{'})$
    set $\mathcal{C} \leftarrow \mathcal{C}_{i,j} := (\mathcal{C} \setminus \{C_i, C_j\}) \bigcup \{C_i \cup C_j\}$
    for *all connected pairs* $C_i, C_k \in \mathcal{C}$ **do**
        if $e_{ik} > 0$ *and* $e_{jk} > 0$ **then**
            $\mathcal{E}_{ik} \leftarrow \mathcal{E}_{ij} + 2m\left(a_k - \mathbb{A}_k - e_{ik} - e_{jk}\right)$
            $e_{ik} \leftarrow e_{ik} + e_{jk}$
            $\Delta_{i,k} \leftarrow \Delta_{i,k} + \Delta_{j,k}$
            delete $\Delta_{j,k}^{'}$
        if $e_{ik} > 0$ *and* $e_{jk} = 0$ **then**
            $\mathcal{E}_{ik} \leftarrow \mathcal{E}_{ij} + 2m\left(a_k - \mathbb{A}_k - e_{ik}\right)$
            $\Delta_{i,k} \leftarrow \Delta_{i,k} - 2a_j a_k$
        if $e_{ik} = 0$ *and* $e_{jk} > 0$ **then**
            $\mathcal{E}_{ik} \leftarrow \mathcal{E}_{ij} + 2m\left(a_k - \mathbb{A}_k - e_{jk}\right)$
            $\Delta_{i,k} \leftarrow \Delta_{j,k} - 2a_i a_k$
            $e_{ik} \leftarrow e_{jk}$
            replace $\Delta_{j,k}^{'}$ with $\Delta_{i,k}^{'}$
        if $\mathcal{E}_{ik} = 0$ **then**
            set $\Delta_{i,k}^{'} \leftarrow \Delta_{i,k}$
        else
            set $\Delta_{i,k}^{'} \leftarrow \frac{\Delta_{i,k}}{\mathcal{E}_{ik}}$
        $\mathbb{A}_i \leftarrow \mathbb{A}_i + \mathbb{A}_j + e_{ij}$
        $a_i \leftarrow a_i + a_j$
return $\mathcal{C}$

5. Experimente

In diesem Kapitel werden experimentell die verschiedenen Prioritätsfunktionen verglichen, die in Kapitel 3 beschrieben wurden. Der CNM-Algorithmus ist mit diesen verschiedenen Prioritätsfunktionen implementiert worden. Für jede Prioritätsfunktion sind folgende Werte bei der Ausführung auf jedem Graphen gespeichert worden: der Wert an Modularity, die Ausführungszeit, der *Mean Weight Balance Factor* des dabei entstehenden Dendrogramms und die Anzahl der aktualisierten Kanten. Wir haben die Anzahl der aktualisierten Kanten gespeichert, weil sie implementierungsunabhängig ist.

Der Mean Weight Balance Factor, abgekürzt MWBF [PB84], wertet aus, wie gut das bei der Ausführung des Algorithmus entstehende Dendrogramm balanciert ist. Für einen gegebenen binären Baum T mit n Knoten wird MWBF(T) wie folgt berechnet: Für einen beliebigen Knoten $u \in T$ bezeichnet n_1 die Anzahl der Knoten im Teilbaum mit Wurzel u, n_2 die Anzahl der Knoten im linken Unterbaum des Knotens u und n_3 die Anzahl der Knoten im rechten Unterbaum des Knotens u. Das heißt, $n \geq n_1 = n_2 + n_3 + 1$. Es gilt, $0 < \text{MWBF(T)} \leq 1$ und

$$\text{MWBF(T)} = \frac{1}{n} \sum_{u \in T} 2 \cdot \frac{1 + \min(n_2, n_3)}{1 + n_1}.$$

Die Experimente sind auf 54 verschiedenen Graphen ausgeführt worden. Wir haben jeden Graphen einer eindeutigen Nummer zugeordnet. Diese Nummerierung ist in den nachfolgenden Tabellen dargestellt. Dabei sind für jeden Graphen auch die Anzahl der Knoten, die Anzahl der Kanten, der Clusterkoeffizient und der Gini Koeffizient der Gradverteilung aufgelistet. Die ersten zwanzig Graphen sind den Webseiten von Newman [vMEJN] und Arenas [vAA] entnommen. Diese sind vor allem soziale Netzwerke, auf denen auf Modularity basierende Experimente oft durchgeführt werden. Die anderen Graphen sind aus der Sammlung von Chris Walshaw [vCW]. Diese bilden eine Benchmark, die benutzt wird, um Algorithmen zur Graphpartitionierung zu vergleichen.

Der Clusterkoeffizient ist ein Maß, um abzuschätzen, wie Knoten untereinander verlinkt sind und wie gut Knoten zusammen in einem Cluster vereinigt werden können. Sei $G = (V, E)$ ein Graph mit $n = |V|$ Knoten. Sei $v \in V$ ein Knoten, $N_v = \{k_1, \ldots, k_l\}$ die Menge der Nachbarn von v und $\overline{m}(N_v)$ die Anzahl der Kanten zwischen den Knoten in N_v. Der Clusterkoeffizient C_v des Knotens v ist gleich der Anzahl der Kanten zwischen den Nachbarn von v geteilt durch die maximale Anzahl der Kanten zwischen den Nachbarn

von v:

$$C_v = \frac{\overline{m}(N_v)}{\binom{l}{2}} = \frac{2\overline{m}(N_v)}{l(l-1)}$$

Der Clusterkoeffizient des Graphen G ist:

$$C_G = \frac{1}{n} \sum_{v \in V} C_v.$$

Der Gini Koeffizient [Gin12] ist ein statistisches Maß für die Ungleichverteilung und wird mithilfe der *Lorenz-Kurve* definiert. Man erhält die Lorenz-Kurve wie folgt: Auf der X-Achse werden Knoten $v_1, \ldots, v_n$ vom kleinsten Grad bis zum größten Grad sortiert ($deg(v_1) \leq deg(v_2) \leq \cdots \leq deg(v_n)$). Für jeden Knoten v_i auf der X-Achse bildet sich ein Punkt (v_i, y_i) der Lorenz-Kurve, wobei

$$y_i = \left(\sum_{j \leq i} deg(v_j) \right) / \left(\sum_{j=1}^{n} deg(v_j) \right).$$

Sei A die Fläche zwischen der Zeile der totalen Gleichverteilung und der Lorenzkurve. B ist die Fläche unter der Lorenz-Kurve (siehe Abbildung 5.1). Der Gini Koeffizient G ist wie folgt definiert:

$$\mathrm{G} = \frac{A}{A+B}.$$

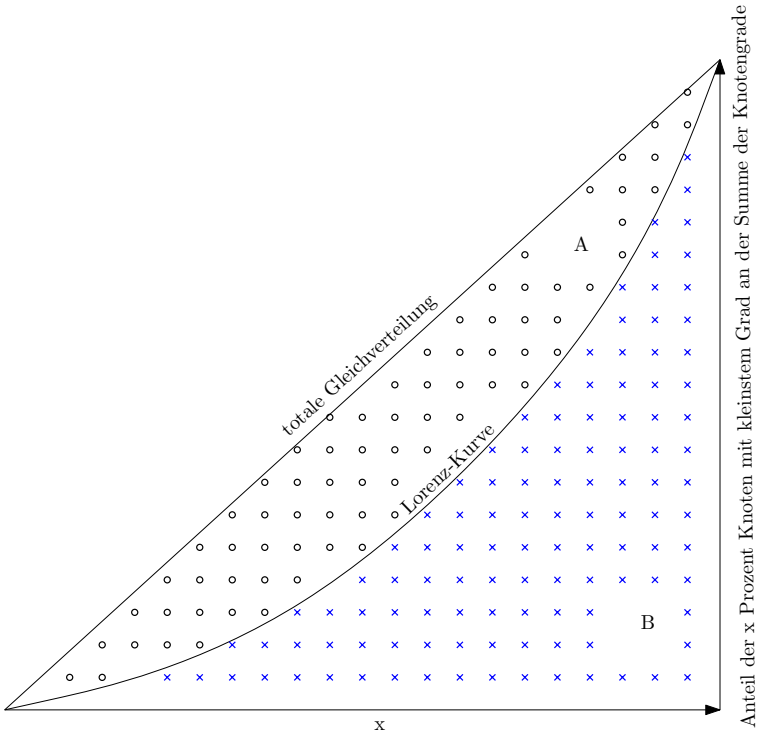

Abbildung 5.1.: Graphische Darstellung des Gini Koeffizienten

Graphen	Nummer	Knoten	Kanten	Clusterkoef.	Gini Koef.
Newman und Arenas Graphen					
adjnoun	1	112	425	0.172840	0.4173
as-22july06	2	22963	48436	0.230448	0.6319
astro-ph	3	16706	121251	0.638800	0.6079
celegans metabolic	4	453	2024	0.645888	0.4951
celegansneural	5	297	2148	0.292363	0.3855
cond-mat-2003	6	31163	120029	0.631500	0.5308
cond-mat-2005	7	40421	175693	0.636200	0.5498
cond-mat	8	16726	47594	0.620400	0.4837
dolphins	9	62	159	0.258958	0.3250
email	10	1133	5451	0.220176	0.4911
football	11	115	613	0.403216	0.0400
hep-th	12	8361	15751	0.442000	0.5118
jazz	13	198	2742	0.617451	0.3460
karate	14	34	77	0.399725	0.3785
lesmis	15	77	254	0.573137	0.4610
netscience	16	1589	2742	0.637800	0.4624
PGPgiantcompo	17	10680	24316	0.265945	0.5918
polblogs	18	1490	16715	0.262652	0.6899
polbooks	19	105	441	0.487527	0.3280
power	20	4941	6594	0.080104	0.3248

Tabelle 5.1.: Graphen 1-20

5.1. Vergleich von verschiedenen Prioritätsfunktionen

Wir vergleichen die Prioritätsfunktionen zuerst auf den Newman und Arenas Graphen und dann auf den Walshaw Graphen. Wir haben bemerkt, dass sich manche Prioritätsfunktionen auf diesen beiden Gruppen von Graphen unterschiedlich verhalten. Deshalb wird für eine bessere Veranschaulichung der Vergleich auf diesen Gruppen von Graphen getrennt ausgeführt. Zuerst vergleichen wir die Prioritätsfunktionen miteinander bezüglich ihrer Werte an Modularity.

Das Schaubild 5.2 illustriert die erzielten Modularity Werte mit verschiedenen Prioritätsfunktionen angewendet auf Newman und Arenas Graphen. Dabei ist auffallend, dass sich die Werte an Modularity mit allen Prioritätsfunktionen ähnlich verhalten. Die Werte der HE-Priorität, HN-Priorität und CNM-Priorität liegen teilweise etwas unter den Werten anderer Prioritätsfunktionen. Ansonsten haben die Prioritätsfunktionen ähnliche Werte an Modularity. Sollte sich der Vergleich der verschiedenen Prioritätsfunktionen bezüglich ihrer Werte an Modularity auf diese Graphen beschränken, wären sie alle fast gleich bewertet.

Auf dem Schaubild 5.3 sind die Werte an Modularity von verschiedenen Prioritätsfunktionen nach der Ausführung auf den Walshaw Graphen zu sehen. Im Vergleich zu Schaubild 5.2 bemerken wir auf diesem Schaubild, dass sich die Werte an Modularity der verschiedenen Prioritätsfunktionen deutlich unterschiedlich verhalten. Zum Beispiel verhält sich die DA-Priorität bezüglich ihrer Werte an Modularity auf diesen Graphen deutlich schlechter als auf den ersten zwanzig Graphen. Die Werte an Modularity der Prioritätsfunktionen HE-Priorität, HN-Priorität, CNM-Priorität und DA-Priorität sind bei diesen Graphen im Allgemeinen deutlich kleiner als die der anderen Prioritätsfunktionen. Was die anderen Prioritätsfunktionen angeht, sind die Werte der Sig-Priorität oft leicht über den anderen Werten. Ansonsten gibt es im Allgemeinen keinen großen Unterschied bezüglich ihrer Werte an Modularity. Die Walshaw Graphen sind viel größer als die Newman und Arenas

Graphen	Nummer	Knoten	Kanten	Clusterkoef.	Gini Koef.
Walshaw Graphen					
144	21	144649	1074393	0.422955	0.1022
3elt	22	4720	13722	0.411055	0.0446
4elt	23	15606	45878	0.407650	0.0340
598a	24	110971	741934	0.425972	0.1243
add20	25	2395	7462	0.638069	0.5901
add32	26	4960	9462	0.615413	0.3853
auto	27	448695	3314611	0.414662	0.1016
bcsstk29	28	13992	302748	0.613806	0.1766
bcsstk30	29	28924	1007284	0.662048	0.2421
bcsstk31	30	35588	572914	0.580956	0.2560
bcsstk32	31	44609	985046	0.595579	0.1781
bcsstk33	32	8738	291583	0.514216	0.1289
brack2	33	62631	366559	0.503743	0.2374
crack	34	10240	30380	0.469696	0.1674
cs4	35	22499	43858	0.056536	0.0234
cti	36	16840	48232	0.004895	0.0368
data	37	2851	15093	0.485719	0.1351
fe 4elt2	38	11143	32818	0.416076	0.0709
fe body	39	45087	323844	0.036488	0.1299
fe ocean	40	143437	409593	0	0.0429
fe pwt	41	36519	278247	0.128347	0.0479
fe rotor	42	99617	662431	0.431419	0.1400
fe sphere	43	16386	49152	0.400098	0.0001
fe tooth	44	78136	452591	0.511325	0.2448
finan512	45	74752	261120	0.503401	0.3145
m14b	46	214765	1679018	0.424820	0.1121
memplus	47	17758	54196	0.765661	0.5470
t60k	48	60005	89440	0	0.0062
uk	49	4824	6837	0.000207	0.0501
vibrobox	50	12328	165250	0.493799	0.3043
wave	51	156317	1059331	0.423494	0.1335
whitaker3	52	9800	28989	0.406027	0.0294
wing	53	62032	121544	0.055595	0.0191
wing nodal	54	10937	75488	0.421302	0.1165

Tabelle 5.2.: Graphen 21-54

Graphen. Wir vermuten, dass sich die einzelnen Prioritätsfunktionen auf dichten Graphen, zum Beispiel den Graphen 27, 29 und 46, deutlich unterscheiden.

Wir vergleichen nun die verschiedenen Prioritätsfunktionen miteinander bezüglich ihrer Laufzeiten. Dafür vergleichen wir die Beschleunigung des Originalalgorithmus mit den verschiedenen Prioritätsfunktionen. Sei Prio eine Prioritätsfunktion. Die Beschleunigung des Originalalgorithmus mit der Prioritätsfunktion Prio bei der Ausführung auf einem Graphen ist gleich:

$$\text{Speedup(Prio)} = \frac{\text{Laufzeit(CNM} - \text{Priorität)}}{\text{Laufzeit(Prio)}}.$$

Auf den Schaubildern 5.4 und 5.5 ist zu sehen, dass die CNM-Priorität meistens deutlich langsamer als die anderen Prioritätsfunktionen ist. Diese anderen Prioritätsfunktionen verschnellern den Originalalgorithmus mit einem Speedup bis ca 30. Abgesehen davon, dass die DA-Priorität bei den Walshaw Graphen generell deutlich langsamer als die anderen Prioritäten ist, verhält sich der Speedup dieser Prioritätsfunktionen bei allen Graphen im Allgemeinen ähnlich. Die Laufzeit der CNM-Priorität ist deutlich schlechter als die Laufzeit anderer Prioritätsfunktionen bei großen Graphen. Bei kleinen oder sehr dünnen Graphen ist der Originalalgorithmus fast genauso schnell wie alle anderen Prioritätsfunktionen. Wir gehen später nochmal auf diese Laufzeiten ein, um sie mit der Anzahl der aktualisierten Kanten und dem Mean Weight Balance Factor zu vergleichen.

Wir schauen uns nun an, wie balanciert das Dendrogramm bei verschiedenen Prioritätsfunktionen ist. Die Schaubilder 5.6 und 5.7 belegen die Unbalanciertheit des CNM-Algorithmus. Die DA-Priorität ist bei den Walshaw Graphen fast genauso unbalanciert wie die CNM-Priorität. Die Prioritätsfunktionen HN-Priorität und WD-Priorität sind oft deutlich balancierter als alle andere Prioritätsfunktionen. Dafür sind diese beiden Prioritätsfunktionen bei den Schaubildern 5.4 und 5.5 zwar schnell aber haben nicht den besten Speedup. Die Prioritätsfunktionen E-Priorität und Sig-Priorität sind etwas schlechter balanciert als die HN-Priorität und die WD-Priorität, haben aber teilweise den besseren Speedup. Dagegen haben die auf dem Schaubild 5.7 nicht gut balancierten Prioritätsfunktionen wie die DA-Priorität und die CNM-Priorität auch auf dem Schaubild 5.5 keine gute Laufzeit. Daraus folgt, dass es keine direkte Korrelation zwischen der Schnelligkeit des Algorithmus und der Balanciertheit des entstehenden Dendrogramms gibt. Durch ein gutes balanciertes Dendrogramm bekommt man eine gute Laufzeitsabschätzung. Aber wenn man durch eine andere Weise versucht, die Laufzeit zu minimieren, kann man eventuell noch eine bessere Laufzeit erzielen.

Wir interpretieren nun die Schaubilder 5.8 und 5.9, die die Anzahl der aktualisierten Kanten zeigen. Dabei bemerken wir, dass es bei der Ausführung der Algorithmen mit verschiedenen Prioritätsfunktionen im Allgemeinen weniger Kanten als im Originalalgorithmus zu aktualisieren gibt. Bei den Walshaw Graphen werden mehr Kanten mit der DA- und CNM-Priorität aktualisiert als mit anderen Prioritätsfunktionen. Die DA-Priorität erzeugt bei den ersten zwanzig Graphen generell gute Ergebnisse im Gegensatz zu den anderen Graphen.

Die Schaubilder 5.8, 5.9, 5.4 und 5.5 sind sich sehr ähnlich. Die Laufzeit der Prioritätsfunktionen hängt sehr von der Anzahl der aktualisierten Kanten ab. Dieses Ergebnis paßt gut zu theoretischen Überlegungen. Das heißt, die Anzahl der bei der Ausführung des Algorithmus auf einem Graphen mit einer Prioritätsfunktion aktualisierten Kanten gibt eine bessere Abschätzung der Laufzeit als der Mean Weight Balance Factor des dabei entstehenden Dendrogramms.

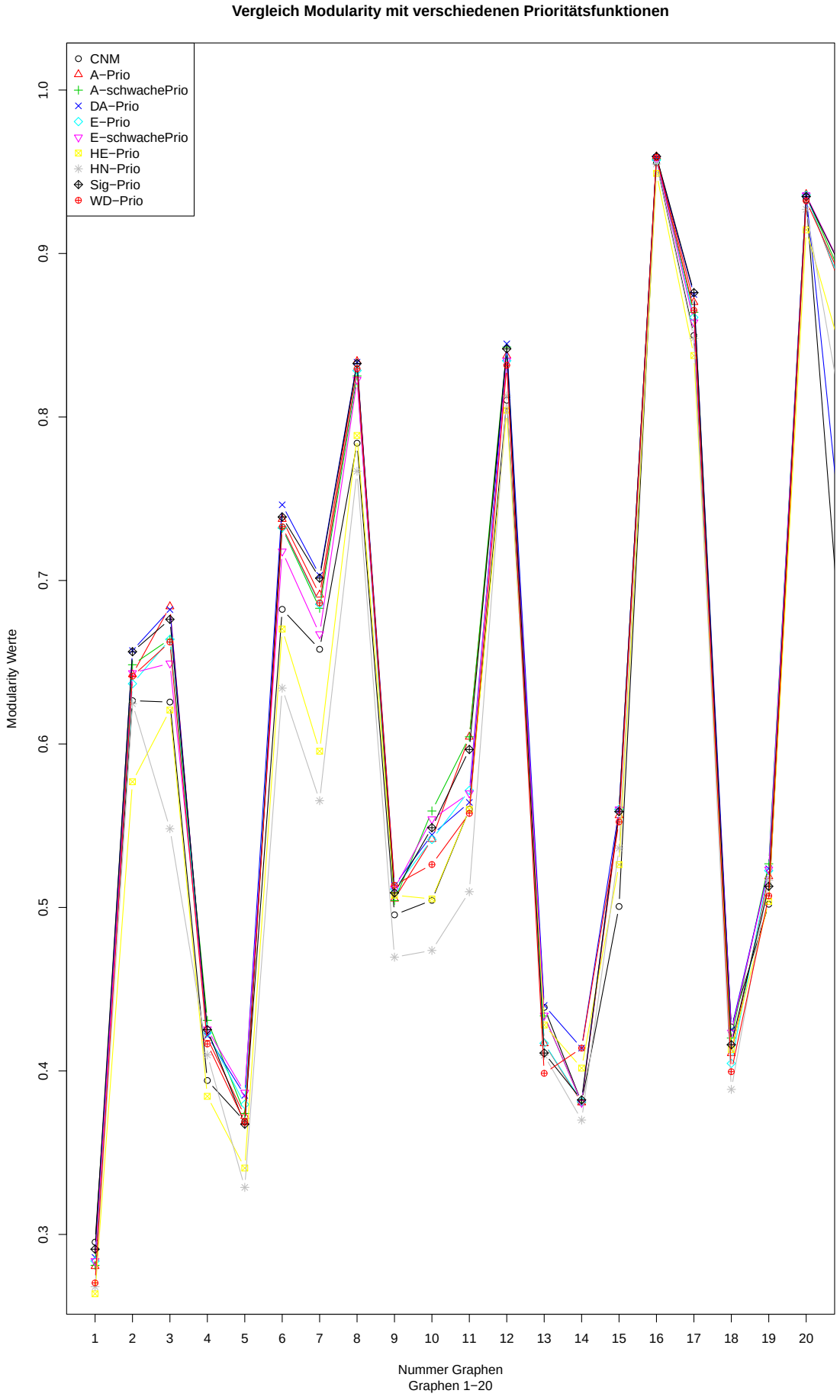

Abbildung 5.2.: Modularity Prioritätsfunktionen

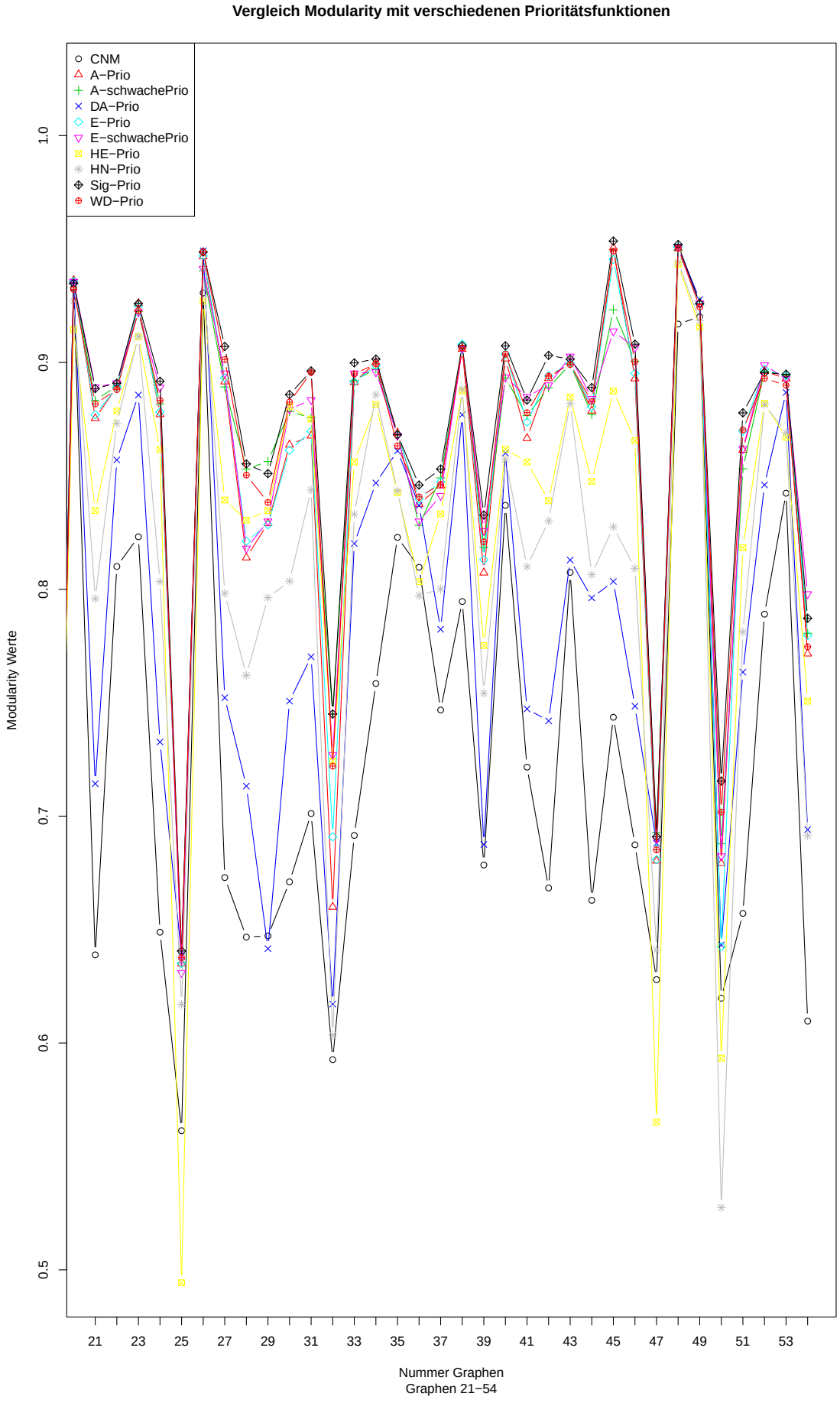

Abbildung 5.3.: Modularity Prioritätsfunktionen

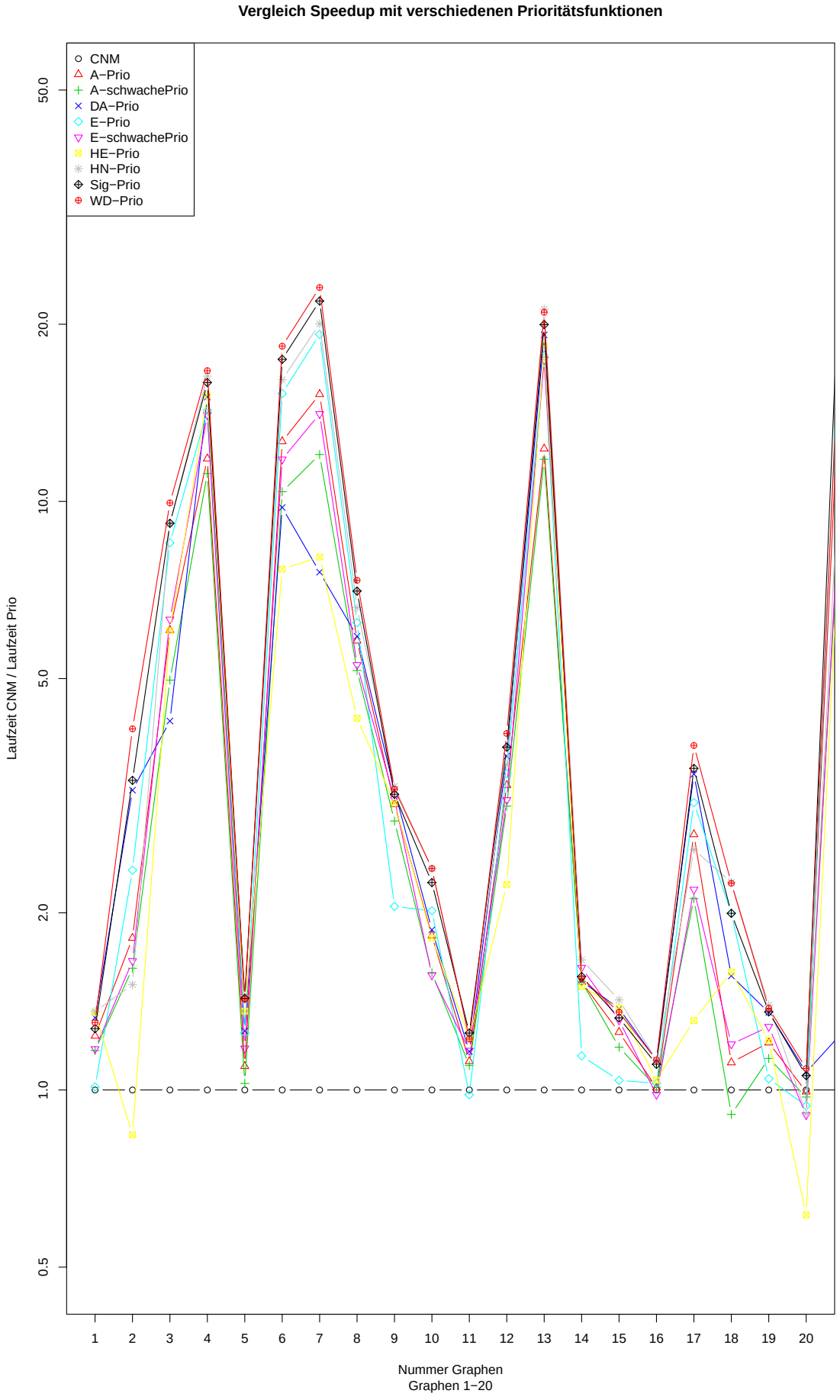

Abbildung 5.4.: Laufzeit Prioritätsfunktionen

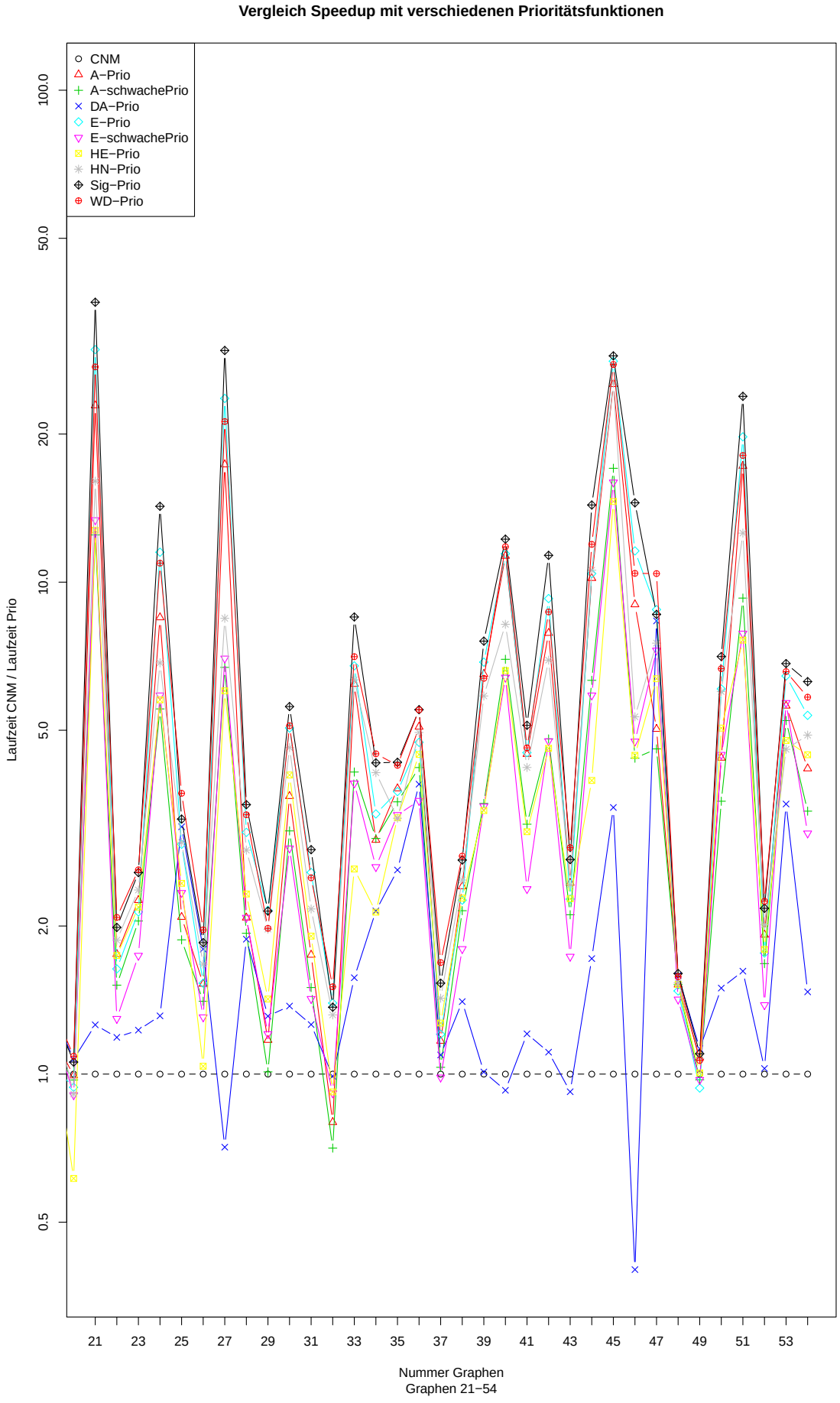

Abbildung 5.5.: Laufzeit Prioritätsfunktionen

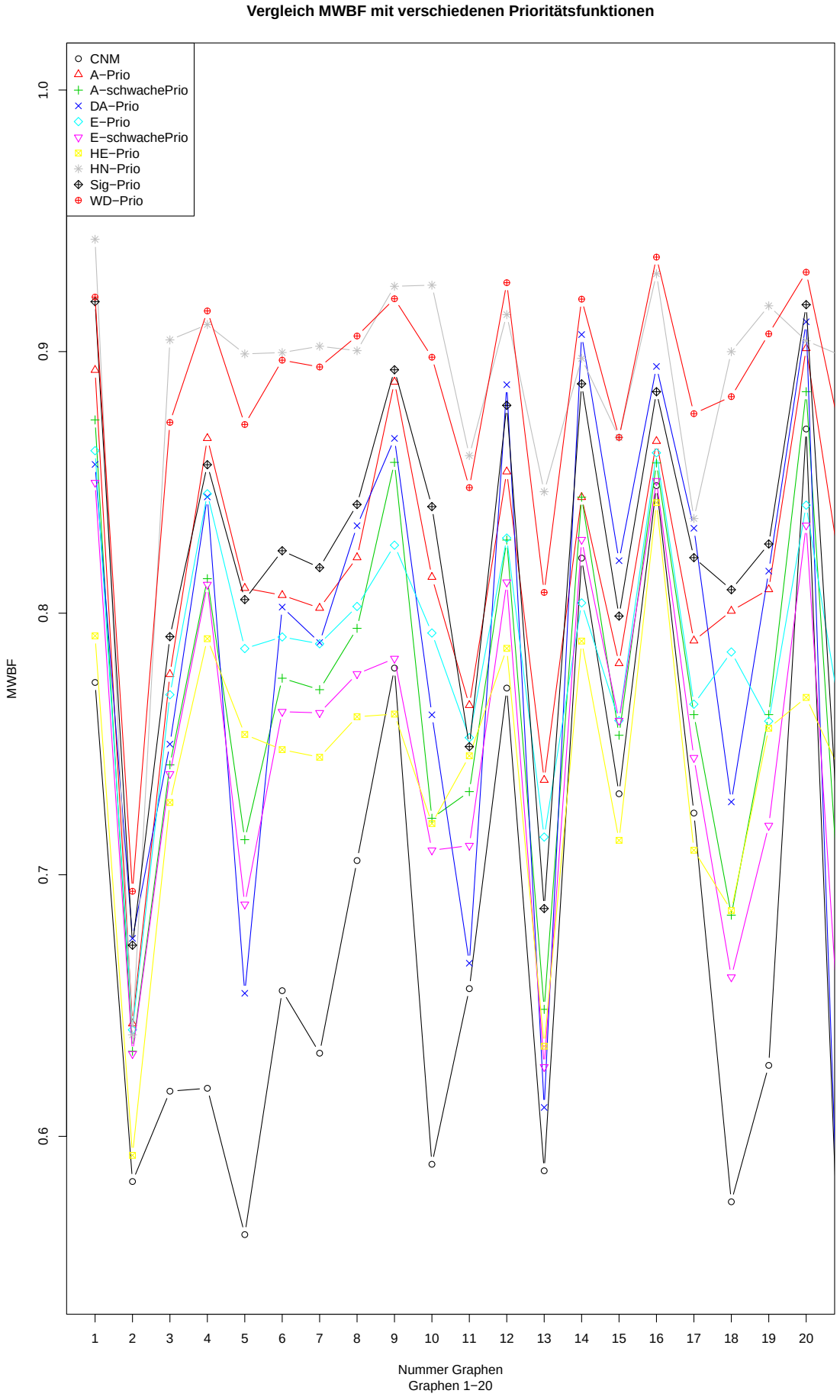

Abbildung 5.6.: Mean Weight Balance Factor Prioritätsfunktionen

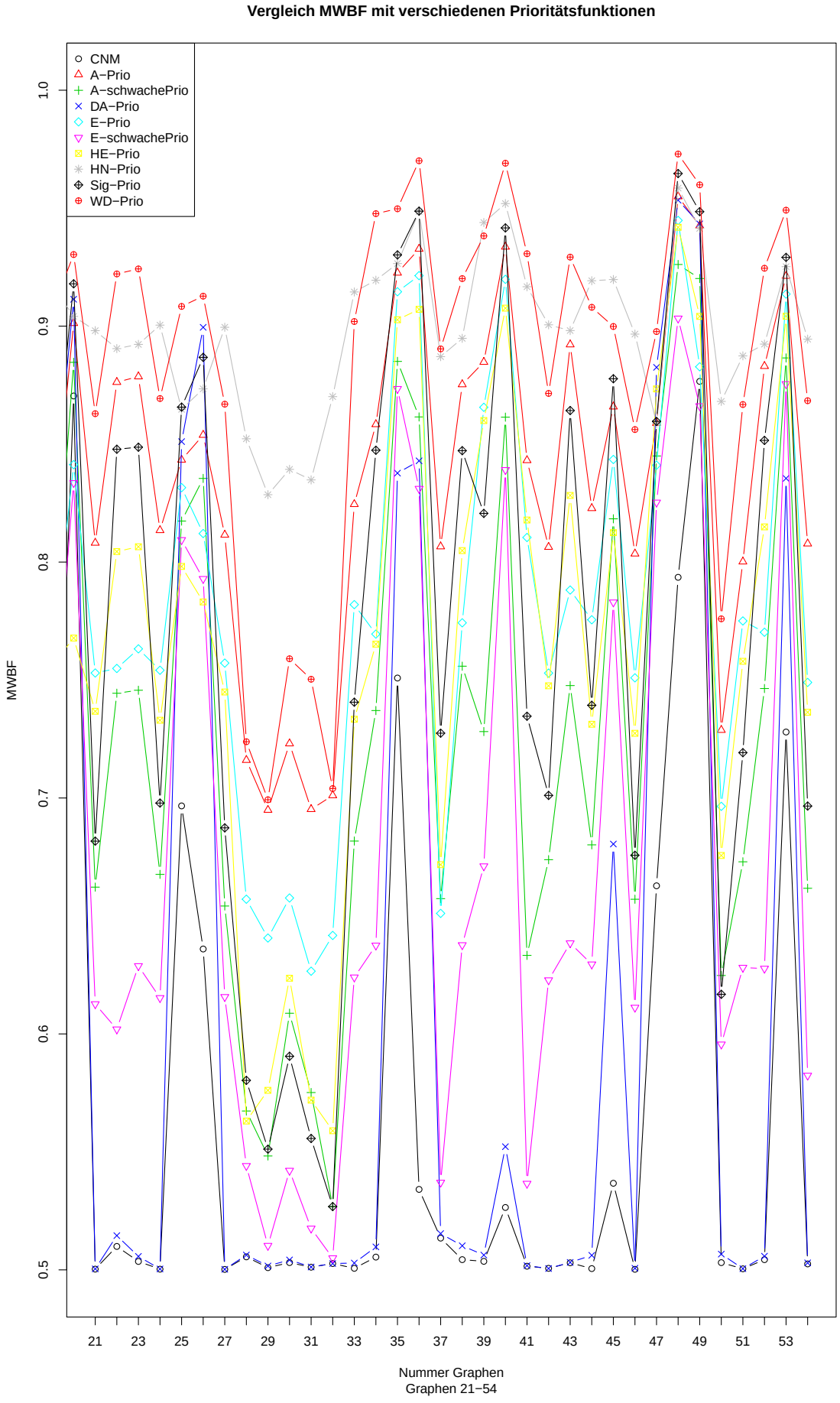

Abbildung 5.7.: Mean Weight Balance Factor Prioritätsfunktionen

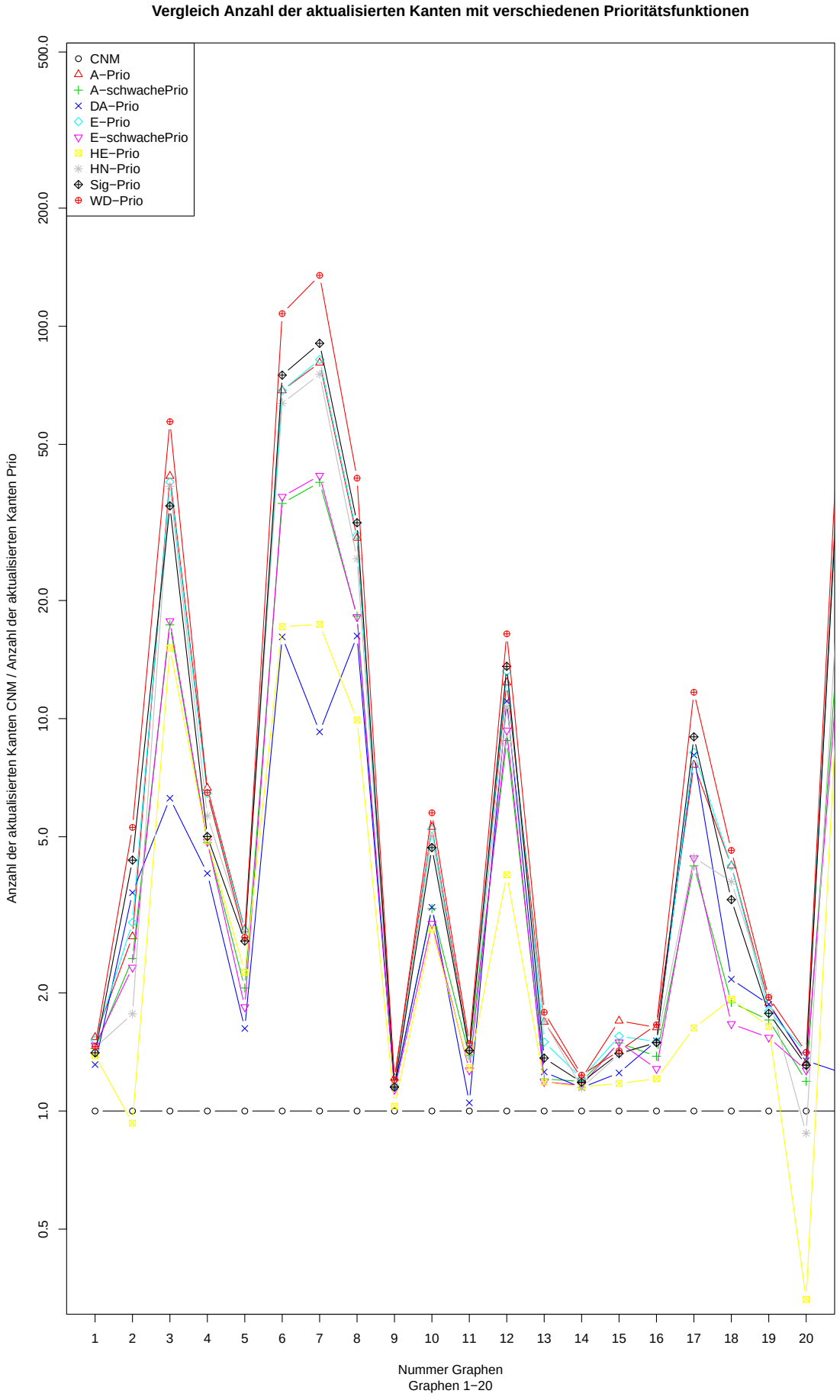

Abbildung 5.8.: Anzahl der aktualisierten Kanten Prioritätsfunktionen

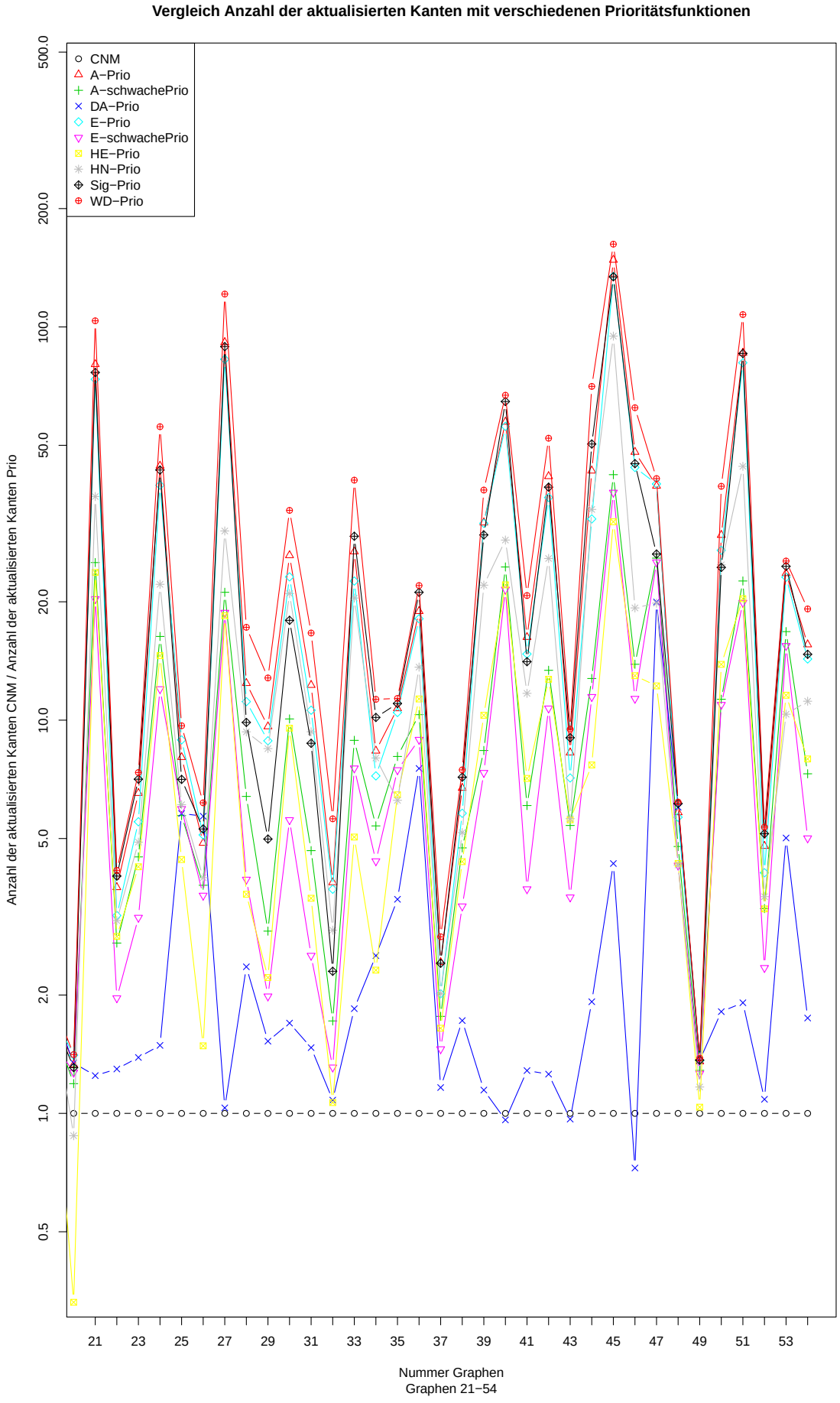

Abbildung 5.9.: Anzahl der aktualisierten Kanten Prioritätsfunktionen

5.2. Auswertungen der Kombinationen

Wir interpretieren nun die Schaubilder für die Kombination der E-Priorität mit der CNM-Priorität. Die anderen Schaubilder sind im Anhang dargestellt. Bei einer Kombination wird für einen gegebenen Graphen mit n Knoten eine balancierte Version des Algorithmus solange ausgeführt, bis $\lceil (1-l) \cdot n \rceil$ Cluster von den n Clustern am Anfang verbleiben. Dann wird der Originalalgorithmus mit den verbleibenden Clustern solange ausgeführt, bis es kein Clusterpaar mehr gibt, dessen Verschmelzen einen Zuwachs an Modularity bringt. Dabei ist $l = 0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, \cdots, 0.95, 1$ ein Tuning Parameter. Das Schaubild einer Kombination besteht aus vier Bildern, auf denen folgende Werte zu sehen sind: der Wert an Modularity, der Mean Weight Balance Factor, die Laufzeit und die Anzahl der aktualisierten Kanten. Jede Spalte ist der Boxplot der Werte bei der Ausführung auf allen 54 Graphen.

Auf dem Schaubild 5.10 der Kombination der E-Priorität mit der CNM-Priorität bemerken wir, dass sich der beste Wert an Modularity ergibt, wenn die E-Priorität und der Originalalgorithmus mit einem Wert von 0.6 des Tuning Parameters kombiniert werden. Das heißt, die Kombination der beiden mit einem gut gewählten Wert des Tuning Parameters bringt einen höheren Wert an Modularity als wenn jede Prioritätsfunktion einzeln benutzt wird. Die Modularity Werte mit den Tuning Parametern 0, 0.6 und 1 lassen sich wie folgt interpretieren: Wenn die E-Priorität und die CNM-Priorität getrennt ausgeführt werden, dann wird mit der E-Priorität ein besserer Wert an Modularity erzielt als mit der CNM-Priorität. Bei der Ausführung des Algorithmus beginnend mit der E-Priorität gibt es auf den meisten Graphen einen Zeitpunkt, ab dem die CNM-Priorität Modularity besser maximieren kann. Dies gilt aber nicht für die DA-Priorität (siehe Schaubild A.3) und die WD-Priorität (siehe Schaubild A.8). Wenn diese Prioritätsfunktionen allein ausgeführt werden, erzeugen sie bessere Werte an Modularity als wenn sie mit dem Originalalgorithmus kombiniert werden. Bei der Kombination mit der HE-Priorität bleiben die Werte an Modularity ab ca. 0.6 etwa gleich (siehe Schaubild A.5). Dagegen werden die Werte an Modularity bei der HN-Priorität ab ca. 0.6 schlechter (siehe Schaubild A.6). Ansonsten verhalten sich die Modularity Werte der Kombination der übrigen Prioritätsfunktionen mit dem Originalalgorithmus ähnlich wie die Kombination mit der E-Priorität. Zusammenfassend lässt sich sagen, dass eine Kombination mit dem Tuning Parameter 0.6 oder größer manchmal einen höheren Wert an Modularity erzielt. Dies ist abhängig von der Prioritätsfunktion.

Die Ergebnisse im linken unteren Bild (Laufzeit) auf dem Schaubild lassen sich mithilfe der Schaubilder 5.4 und 5.5 interpretieren. Die E-Priorität ist deutlich schneller als die CNM-Priorität. Deshalb ist zu erwarten, dass bei der Kombination der beiden Prioritätsfunktionen gilt, dass je länger die E-Priorität auf einem Graphen ausgeführt wird, desto schneller ist die Ausführung. Diese Interpretation gilt auch für die Anzahl der aktualisierten Kanten auf dem Bild rechts unten. Dabei macht auch die Ähnlichkeit der beiden unteren Bilder deutlich, dass Folgendes gilt: die Schnelligkeit einer Prioritätsfunktion hängt mit der Anzahl der aktualisierten Kanten zusammen. Auf dem rechten oberen Bild wird wieder illustriert, dass die E-Priorität balancierter als die CNM-Priorität arbeitet. Bei der Kombination von beiden Prioritätsfunktionen werden die entstehenden Dendrogramme besser balanciert, je länger die E-Priorität ausgeführt wird. Dies gilt auch für alle anderen Kombinationen mit der CNM-Priorität. Entgegen der Vermutung, dass ab irgendeinem Schritt eine Prioritätsfunktion den Algorithmus genug balanciert hat, damit der CNM-Algorithmus balanciert weiterarbeiten kann, beobachten wir, dass dieses Phänomen nicht auftritt. Der CNM-Algorithmus arbeitet weiterhin unbalanciert und langsam. Das heißt, die Kombination einer Prioritätsfunktion mit dem Originalalgorithmus bringt höchstens einen Gewinn bei Modularity, falls der Tuning Parameter gut gewählt ist.

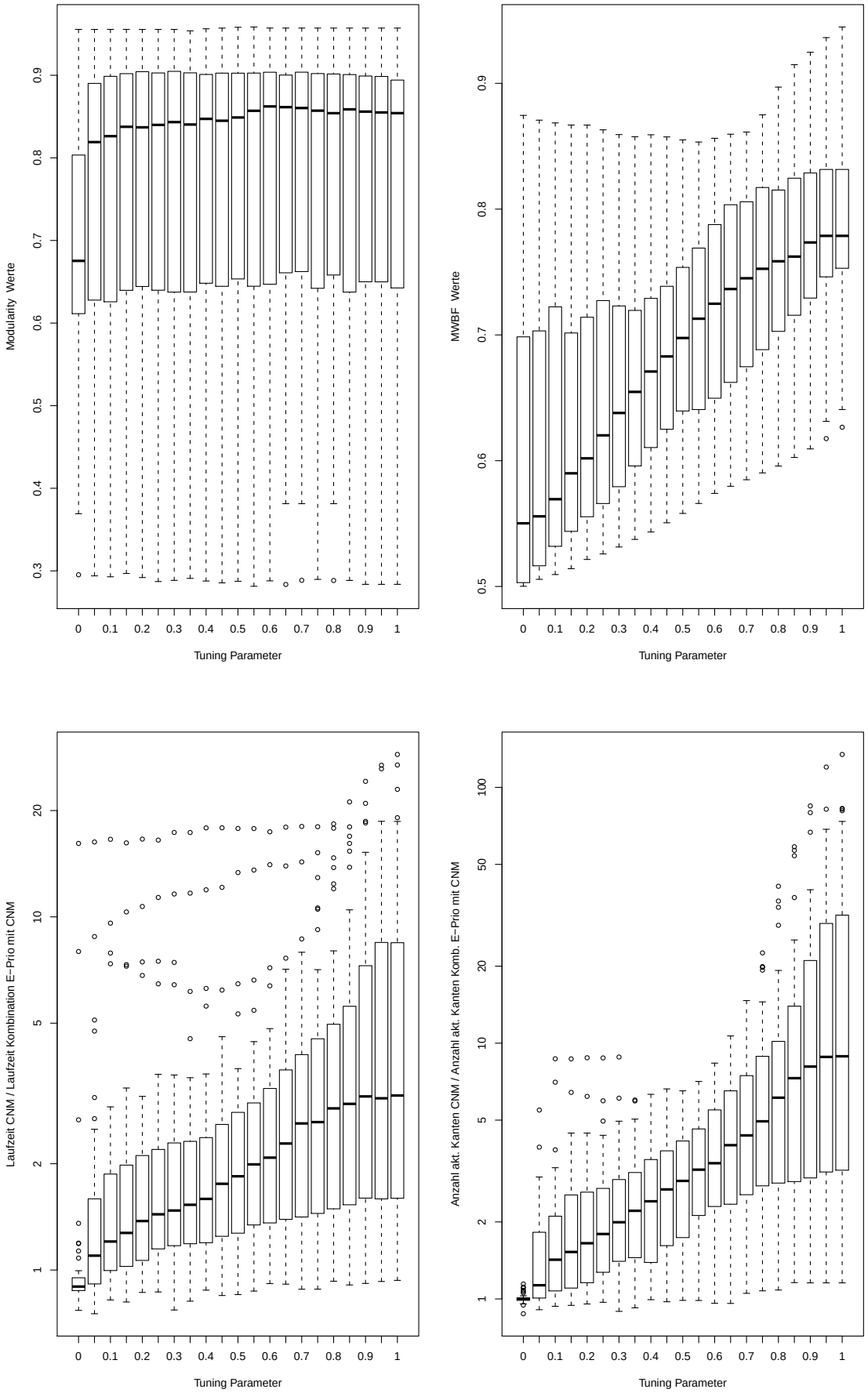

Abbildung 5.10.: Kombination E-Priorität mit CNM

6. Zusammenfassung

Wir haben uns in dieser Arbeit mit dem Graphenclustern beschäftigt. Das Ziel ist dabei gewesen, Graphen so zu clustern, dass die Anzahl der entstehenden Cluster beliebig ist, jedes Cluster eine beliebige Anzahl an Knoten enthält und die Cluster knotendisjunkt sind. Die Qualität der entstehenden Clusterung eines Graphen ist mit der Qualitätsfunktion Modularity bewertet worden. Um gute Modularity Werte zu erzielen, haben wir den CNM-Algorithmus erweitert. Der CNM-Algorithmus ist ein Greedy-Algorithmus, der am Anfang jeden einzelnen Knoten als Cluster betrachtet und iterativ zwei Cluster vereinigt, deren Vereinigung den größten Zuwachs an Modularity bringt.

Ein Dendrogramm ist ein binärer Baum, der die durchgeführten Vereinigungsoperationen repräsentiert. Bei der Ausführung des CNM-Algorithmus auf einem Graphen ist das entstehende Dendrogramm unbalanciert. Das wirkt sich negativ auf die Laufzeit des Algorithmus aus. Um den Algorithmus zu verschnellern, haben wir den CNM-Algorithmus so modifiziert, dass das bei der Ausführung mit der modifizierten Version entstehende Dendrogramm balanciert ist. Für eine gute Qualität sollte diese modifizierte Version auch ähnlich gute Werte an Modularity erzeugen. Dafür haben wir im Originalalgorithmus die Prioritätsfunktion ersetzt. Die Prioritätsfunktion entscheidet in jedem Schritt des Algorithmus darüber, welches Clusterpaar vereinigt wird. Mit der CNM-Priorität wird in jedem Schritt des Algorithmus das Clusterpaar vereinigt, dessen Verschmelzen den größten Zuwachs an Modularity bringt. Wir haben im Algorithmus die CNM-Priorität durch eigene Prioritätsfunktionen ersetzt. Mit der E-Priorität werden in jedem Schritt des Algorithmus Clusterpaare mit einem großen Zuwachs an Modularity und wenigen ausgehenden Kanten vereinigt. Mit der A-Priorität werden bei der Vereinigung von Clustern Clusterpaare mit einer kleinen Anzahl der zu ihnen adjazenten Cluster bevorzugt. Die E-schwache Priorität ist ein Kompromiss zwischen dem Originalalgorithmus und der E-Priorität. Analog ist die A-schwache Priorität ein Kompromiss zwischen dem Originalalgorithmus und der A-Priorität. Mit diesen Prioritätsfunktionen haben wir den Originalalgorithmus gut balanciert, bessere Laufzeiten und sogar teilweise deutlich bessere Modularity Werte erzielt. Dann haben wir diese Prioritätsfunktionen mit anderen Prioritätsfunktionen aus der Literatur verglichen. Unsere Prioritätsfunktionen haben dabei zwar nicht besser, aber fast genauso gut wie die besten Funktionen aus der Literatur funktioniert.

Die Experimente sind auf den Graphen aus dem Walshaw Benchmark und Graphen aus den Webseiten von Newman und Arenas durchgeführt worden. Dabei haben wir bemerkt, dass sich manche Prioritätsfunktionen auf diesen beiden Gruppen von Graphen sehr unterschiedlich verhalten.

Wir haben bei den Implementierungen der verschiedenen Versionen des Algorithmus die Anzahl der aktualisierten Kanten und den Mean Weight Balance Factor (MWBF) als Maß für die Balanciertheit des Dendrogramms gespeichert und mit der Laufzeit verglichen. Bei den Experimenten haben wir beobachtet, dass beides uns eine gute Abschätzung der Laufzeit gibt, die Anzahl der aktualisierten Kanten aber deutlich direkter mit der Laufzeit zusammenhängt. Daraus folgt, dass die Laufzeit nicht immer direkt am Mean Weight Balance Factor (MWBF) festgemacht werden kann.

Weiterhin haben wir uns gefragt, ob die CNM-Priorität vor allem am Anfang unbalanciert arbeitet. Dazu haben wir verschiedene balancierte Prioritätsfunktionen mit der CNM-Priorität kombiniert. Bei einer Kombination wird der Algorithmus mit einer balancierten Prioritätsfunktion bis zu einem bestimmten Zeitpunkt ausgeführt. Dieser Zeitpunkt wird von einem Tuning Parameter bestimmt. Danach wird der Algorithmus mit der CNM-Priorität weiter ausgeführt. Diese Heuristik hat dem Originalalgorithmus nicht dabei geholfen, balanciert zu arbeiten. Dagegen liefert eine gute Wahl des Tuning Parameters bei den meisten Kombinationen einen leichten Gewinn an Modularity verglichen mit der ausschließlichen Benutzung der einzelnen Prioritätsfunktionen.

A. Anhang

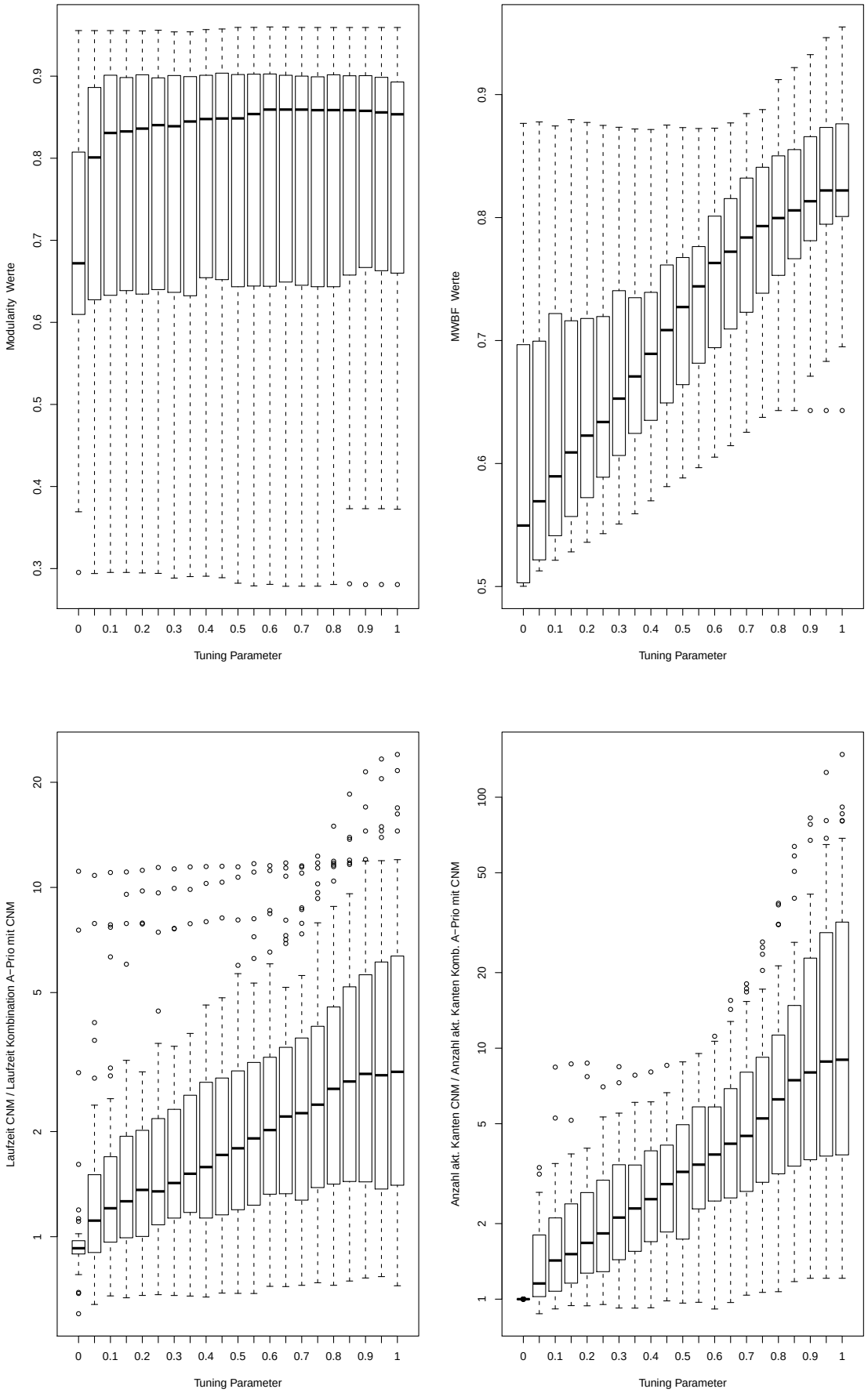

Abbildung A.1.: Kombination A-Priorität mit CNM

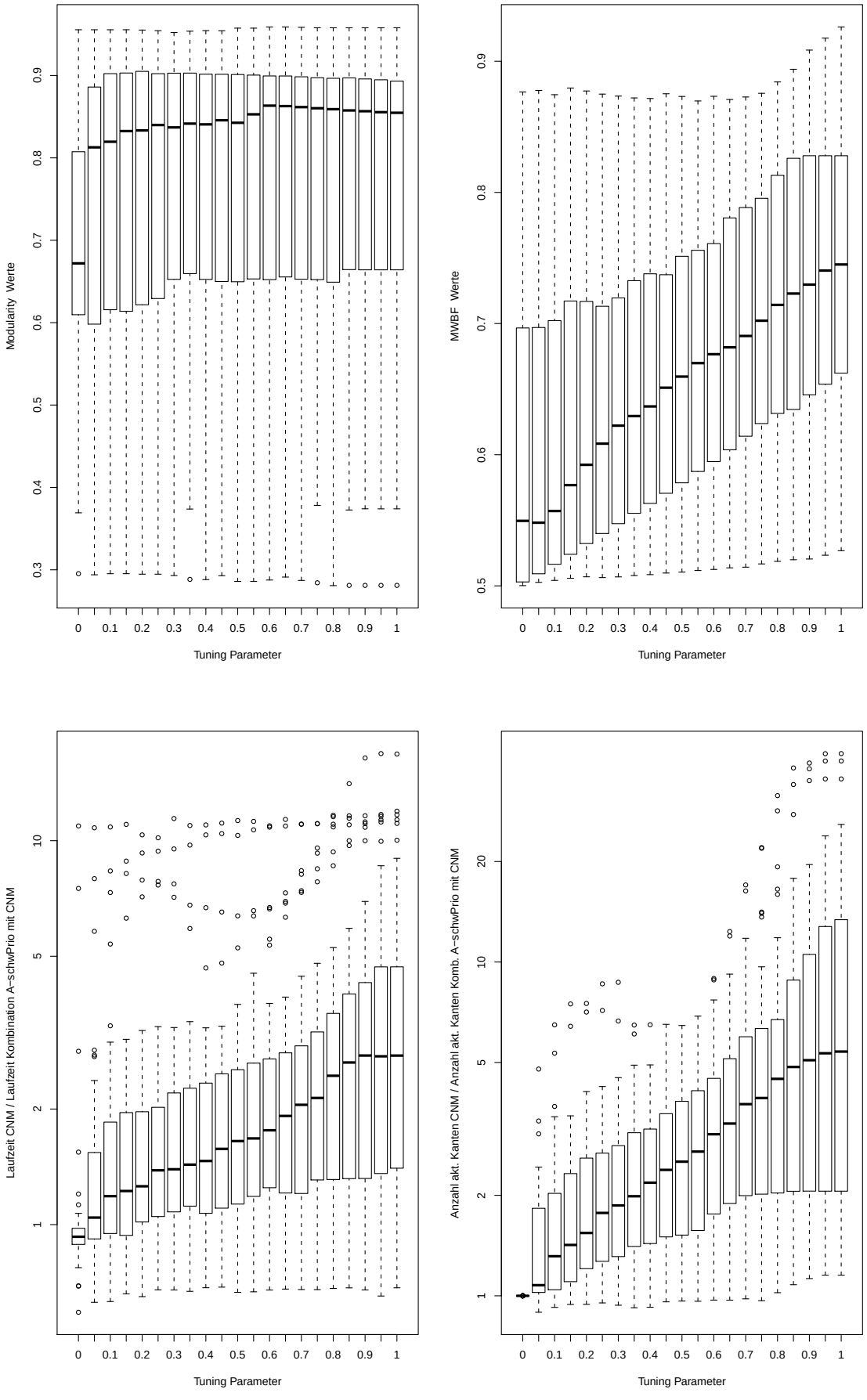

Abbildung A.2.: Kombination A-schwachePriorität mit CNM

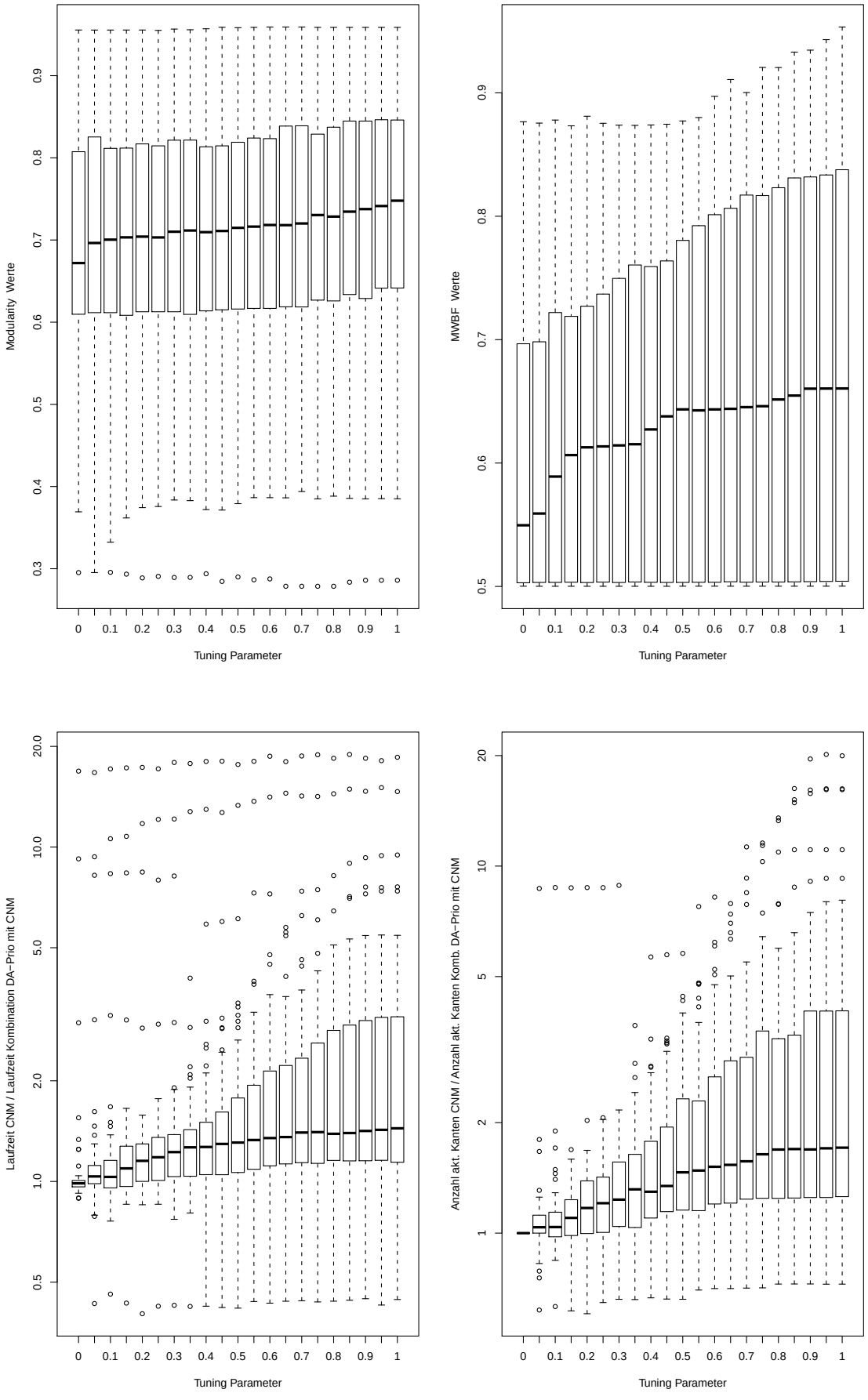

Abbildung A.3.: Kombination DA-Priorität mit CNM

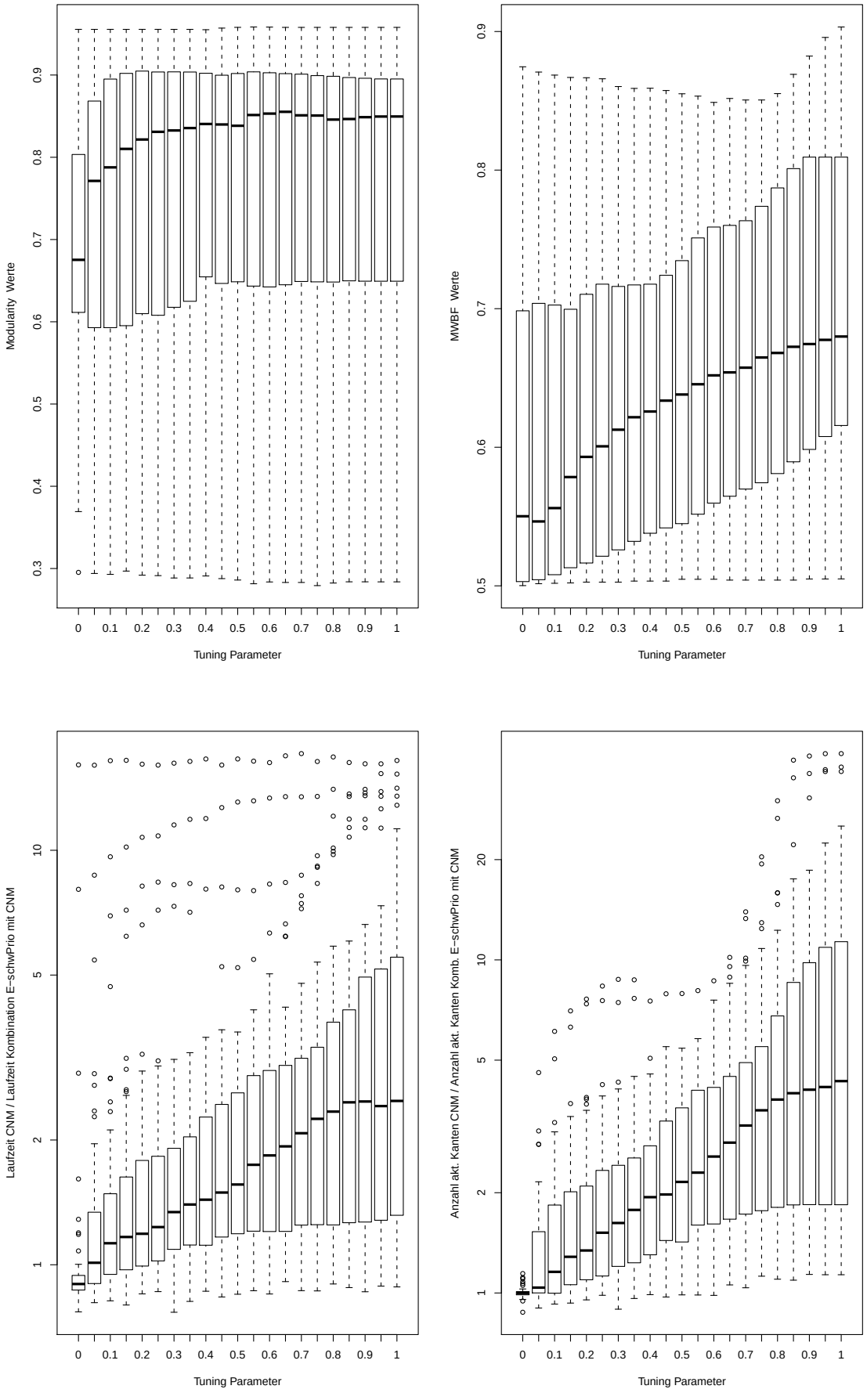

Abbildung A.4.: Kombination E-schwachePriorität mit CNM

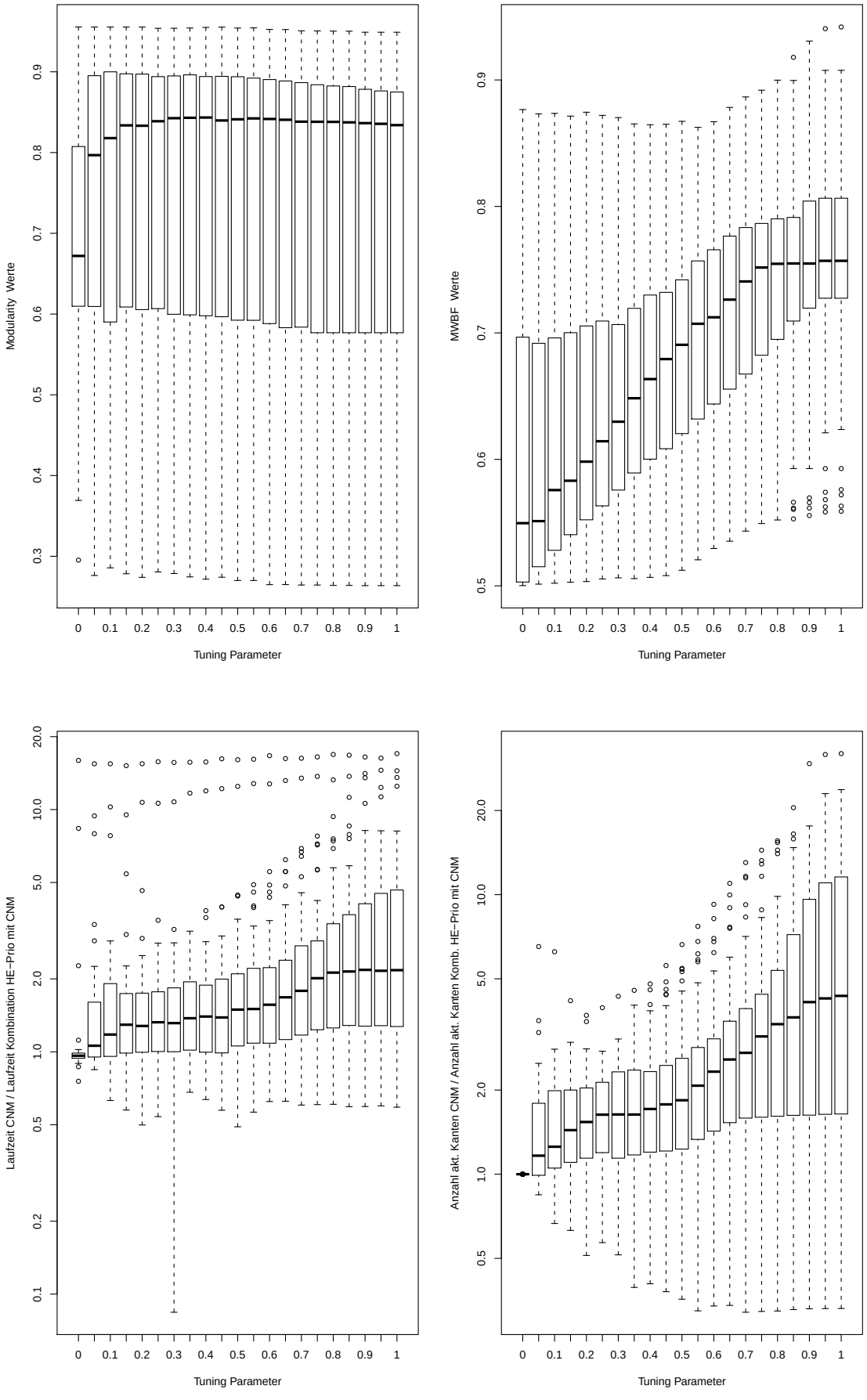

Abbildung A.5.: Kombination HE-Priorität mit CNM

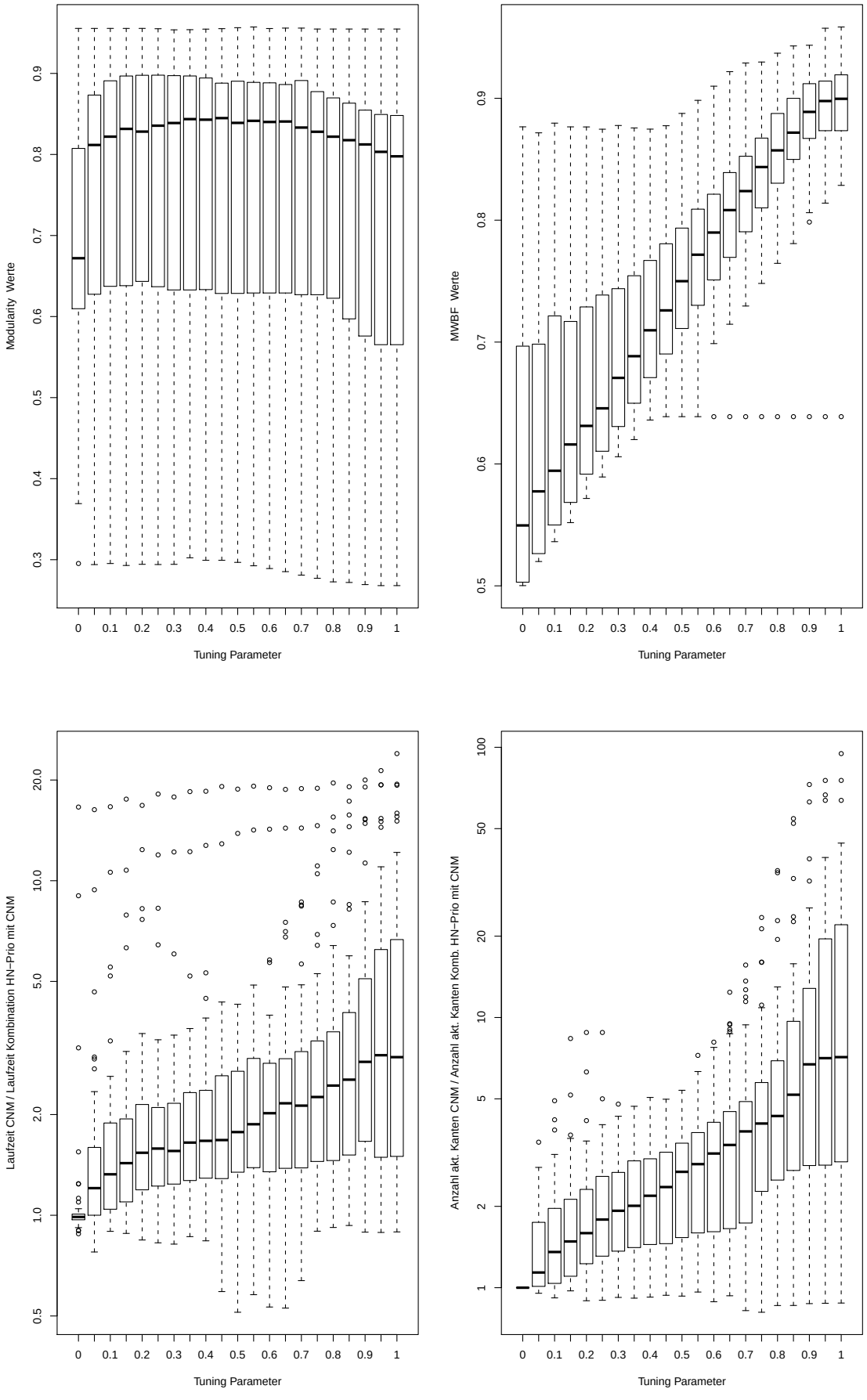

Abbildung A.6.: Kombination HN-Priorität mit CNM

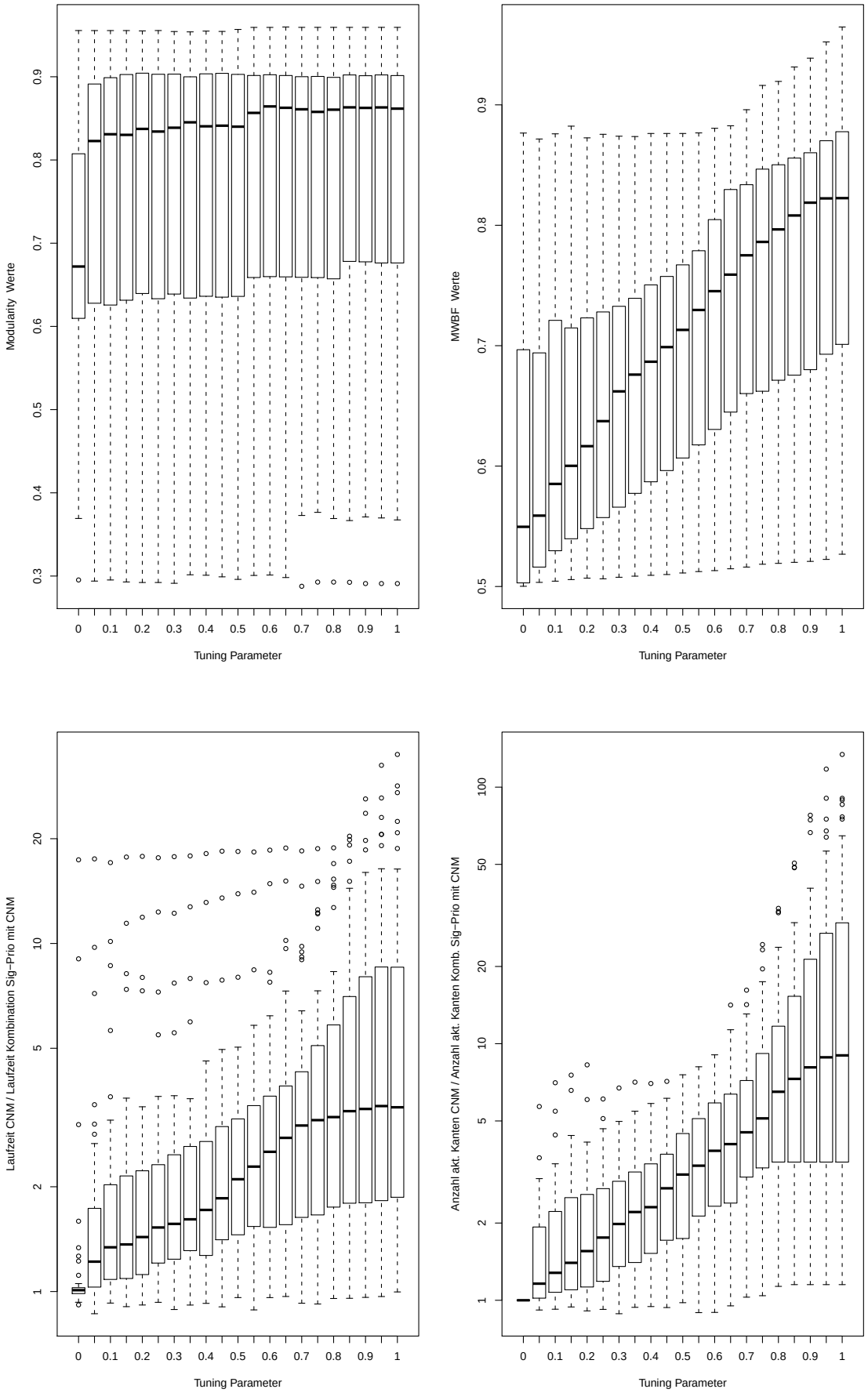

Abbildung A.7.: Kombination Sig-Priorität mit CNM

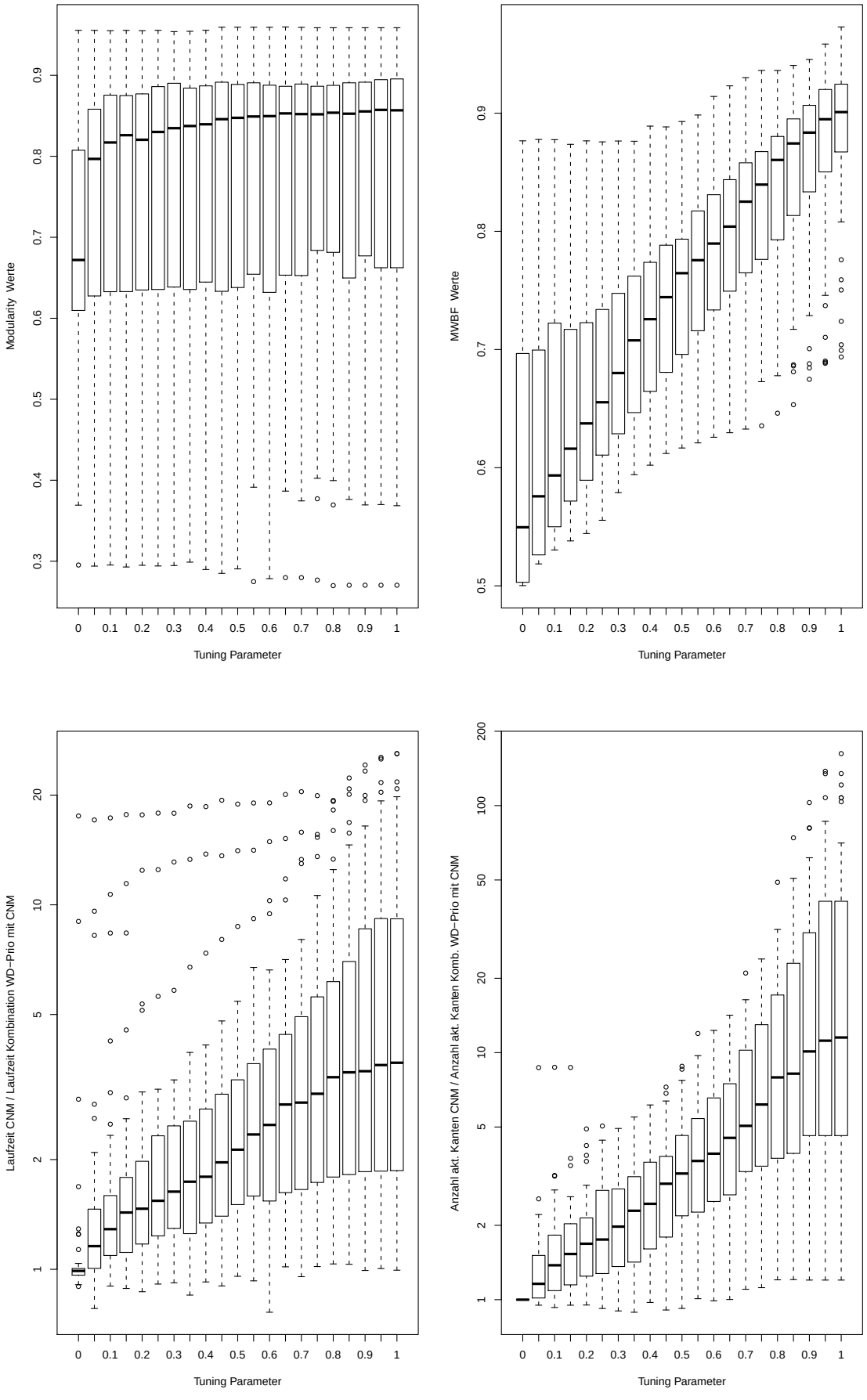

Abbildung A.8.: Kombination WD-Priorität mit CNM

Literaturverzeichnis

[BDH+07] U. Brandes, D. Delling, M. Höfer, M. Gaertler, R. Görke, Z. Nikoloski, and D. Wagner, "On Finding Graph Clusterings with Maximum Modularity," in *Proceedings of the 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG'07)*, ser. Lecture Notes in Computer Science, A. Brandstädt, D. Kratsch, and H. Müller, Eds., vol. 4769. Springer, October 2007, pp. 121–132.

[CNM04] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, no. 066111, 2004. [Online]. Available: http://link.aps.org/abstract/PRE/v70/e066111

[DDGA06] L. Danon, A. Díaz-Guilera, and A. Arenas, "The effect of size heterogeneity on community identification in complex networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2006, no. 11, p. P11010, 2006. [Online]. Available: http://stacks.iop.org/1742-5468/2006/P11010

[Gin12] C. W. Gini, "Variability and mutability, contribution to the study of statistical distributions and relations," *Studi Economico-Giuridici della R. Universita de Cagliari*, 1912, reviewed in: Light, R.J., Margolin, B.H.: An Analysis of Variance for Categorical Data. *J. American Statistical Association*, Vol. 66 pp. 534-544 (1971).

[NR09] A. Noack and R. Rotta, "Multi-level algorithms for modularity clustering," in *SEA*, 2009, pp. 257–268.

[PB84] A. K. Pal and A. Bagchi, "On the mean weight balance factor of binary trees," in *FSTTCS*, 1984, pp. 419–434.

[vAA] W. von Alex Arenas, "http://deim.urv.cat/~aarenas/data/welcome.htm."

[vCW] W. von Chris Walshaw, "http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/."

[vMEJN] W. von Mark E. J. Newman, "http://www-personal.umich.edu/~mejn/netdata/."

[WT07] K. Wakita and T. Tsurumi, "Finding Community Structure in Mega-scale Social Networks," February 2007, technical Report on arXiv. [Online]. Available: http://arxiv.org/abs/cs/0702048v1